68 MICRO JOURNAL

Australia A $8.50 New Zealand NZ $9.85
Singapore S $14.95 Hong Kong H $29.00
Malaysia M $14.25 Sweden 30:-SEK

$2.95USA

OS-9 Systems Integration

6800 6809 68008 68000 68010 68020 68030

The Magazine for Motorola CPU Devices *For Over a Decade!*

This Issue:

A User Contributor Journal

And Lots More!

VOLUME XI ISSUE VII ● Devoted to the 68XXX User ● Aug./Sept. 1989

The Grandfather of "DeskTop Publishing™"

SERVING THE 68XXX USER WORLDWIDE

0 09
74470 12810

PHOTO CREDIT: NASA

Mustang-020 Mustang-08 Benchmarks

	32 bit Integer	Register Long
IBM AT 7300 Xenix Sys 3	9.7	
AT&T 7300 UNIX PC 68010	7.2	4.3
DEC VAX 11/780 UNIX Berkley 4.2	3.6	3.2
DEC VAX 11/750 " "	5.1	3.2
68008 OS-9 68K 8 Mhz	18.0	9.0
68000 OS-9 68K 10 Mhz	6.5	4.0
MUSTANG-08 68008 OS-9 68K 10 Mhz	***9.8***	***6.3***
MUSTANG-020 68020 OS-9 68K 16 Mhz	***2.3***	***0.88***
MUSTANG-020 68020 MC68881 UniFLEX 16 Mhz	***1.8***	***1.22***

```
Main()
        {
                register long i;
                for (i=0; i < 999999; ++i);
        }
```

Estimated MIPS - MUSTANG-020 ..... 4.5 MIPS,

Burst to 8 - 10 MIPS; Motorola Specs

Mustang Specifications

12.5 Mhz (optional 16.6 Mhz available) MC68020 full 32-bit wide path
32-bit wide data and address buses, non-multiplexed
on chip instruction cache
object code compatible with all 68XXX family processors
enhanced instruction set - math co-processor interface
68881 math hi-speed floating point co-processor (optional)
direct extension of full 68020 instruction set
full support IEEE P754, draft 10.0
transcendental and other scientific math functions
2 Megabyte of SIP RAM (512 x 32 bit organization)
up to 256K bytes of EPROM (64 x 32 bits)
4 Asynchronous serial I/O ports standard
optional to 20 serial ports
standard RS-232 interface
optional network interface
buffered 8 bit parallel port (1/2 MC68230)
Centronics type pinout
expansion connector for I/O devices
16 bit data path
256 byte address space
2 interrupt inputs
clock and control signals
Motorola I/O Channel Modules
time of day clock/calendar w/battery backup
controller for 2, 5 1/4" floppy disk drives
single or double side, single or double density
35 to 80 track selectable (48.96 TPI)
SASI interface
programmable periodic interrupt generator
interrupt rate from micro-seconds to seconds
highly accurate time base (5 PPM)
5 bit sense switch, readable by the CPU
Hardware single-step capability

Don't be mislead! ONLY Data-Comp delivers the Super MUSTANG-020

OS-9

OS-9 Professional Ver *Includes C Compiler	$850.00
Basic09	450.00
C Compiler	500.00
68000 Disassembler (w/source add: $100.00)	100.00
Fortran 77	750.00
Microware Pascal	500.00
Onegaard Pascal	900.00
Stylo-Graph	495.00
Stylo-Spell	195.00
Stylo-Merge	175.00
Stylo Graph-Spell-Merge	695.00
PAT w/C source	229.00
JUST w/C source	79.95
PAT/JUST Combo	249.50
Sculptor+ (see below)	995.00
COM	125.00

UniFLEX

UniFLEX (68020 ver)	$450.00
Screen Editor	150.00
Sort-Merge	200.00
BASIC/PreCompiler	300.00
C Compiler	350.00
COBOL	750.00
CMODEM w/source	100.00
TMODEM w/source	100.00
X-TALK (see Ad)	99.95
Cross Assembler	50.00
Fortran 77	450.00
Sculptor+ (see below)	995.00

Standard MUSTANG-020™ shipped 12.5 Mhz.

Add for 16.6 Mhz 68020	375.00
Add for 16.6 Mhz 68881	375.00
Add for 20 Mhz 68020/RAM	750.00
16 Port exp. RS-232	335.00
Requires 1 or 2 Adapter Cards below RS232 Adapter	165.00

Each card supports 4 additional ser. ports (total of 36 serial ports supported)

60 line Parallel I/O card	398.00

Uses 3 68230 Interface/Timer chips, 6 groups of 8 lines each, separate buffer direction control for each group.

Prototype Board	75.00

area for both dip and PGA devices & a pre-wired memory area up to 512K DRAM.

SBC-AN	475.00

Interface between the system and ARCNET modified token-passing LAN, fiber optics optional - call. LAN software drivers 120.00

Expansion for Motorola I/O Channel Modules	$195.00

Special for complete MUSTANG-020™ system buyers - Sculptor+ $695.00. SAVE $300.00

Software Discounts

All MUSTANG-020™ system and board buyers are entitled to discounts on all listed software: 10-70% depending on item. Call or write for quotes. Discounts apply after the sale as well.

The PRO!

These hi-speed 68020 systems are presently working at NASA, Atomic Energy Commission, Government Agencies as well as Universities, Business, Labs, and other Critical Applications Centers, worldwide, where speed, math crunching and multi-user, multi-tasking UNIX C level V compatability and low cost is a must.

Only the "PRO" Version of OS-9 Supported!

This is *HEAVY DUTY* Country!

For a limited time we will offer a $400 trade-in on your old 68XXX SBC. Must be working properly and complete with all software, cables and documentation. Call for more information

Price List:

Mustang-020 SBC		$2490.00
Cabinet w/switching PS		$299.95
5"-80 track floppy DS/DD		$269.95
Floppy Cable		$39.95
OS-9 68K Professional Version		$850.00
C Compiler ($500 Value)		N/C
Winchester Cable		$39.95
Winchester Drive 25 Mbyte		$895.00
Hard Disk Controller		$395.00
Shipping USA UPS		$20.00
UniFLEX	Less	$100.00
MC68881 f/p math processor	Add	$275.00
16.67 Mhz MC68020		$375.00
16.67 Mhz MC68881		$375.00
20 Mhz MC68020 Sys		$750.00
Note all 68881 chips work with 20 Mhz Sys		
	Total:	$5299.80

NEW LOWER PRICES

25 Mbyte HD ~~$4299.80~~ $3749.80

85 Mbyte HD ~~$5148.80~~ $4548.80

Note: Only Professional OS-9 Now Available (68020 Version) Includes ($500) C Compiler - 68020 & 68881 Supported - For UPGRADES Write or Call for Professional OS-9 Upgrade Kit

Data-Comp Division

"A Decade of Quality Service"

Systems World-Wide

Computer Publishing, Inc. 5900 Cassandra Smith Road

Hixson, Tn 37343

Telephone (615) 842-4601 - FAX (615) 842-7990

A Member of the CPI Family

68 Micro Journal

10 Years of Dedication to Motorola CPU Users

The Originator of "DeskTop Publishing™"

6800 6809 68000 68010 68020

Publisher
Don Williams Sr.

Executive Editor
Larry Williams

Production Manager
Tom Williams

Office Manager
Joyce Williams

Subscriptions
Carolyn Williams

Contributing & Associate Editors

Ron Anderson — Dr. E.M. "Bud" Pass
Ron Voigts — Art Weller
Doug Lurie — Dr. Theo Elbert

Toll Free Subscription Line 1-800 669 6809

Subscription Rates

USA
1 Year $ 24.50, 2 years $ 42.50, 3 years $ 64.50
Canada & Mexico, USA funds
1 Year $ 34.00, 2 Years $ 61.50, 3 Years $ 93.00
Other Countries
Surface Rates , USA funds
1 Year $ 36.50, 2 Years $ 66.50, 3 Years $ 100.50
Air Mail Rates, USA funds
1 Year $ 72.50, 2 Years $ 138.50, 3 Years $ 208.50

COMPUTER PUBLISHING, INC.
"Over a Decade of Service"
"World Wide"

68 MICRO JOURNAL
Computer Publishing Center
5900 Cassandra Smith Road
PO Box 849
Hixson, TN 37343
Phone (615) 842-4600 FAX (615) 842-7990

68 Micro Journal is the *original* "DeskTop Publishing" product and has continously published since 1978 using only micro-computers and special "DeskTop" software. Using first a kit built 6800 micro-computer, a modified "ball" typewriter, and "home grown" DeskTop Publishing software. None was commercially available at that time. For over 10 years we have been doing "DeskTop Publishing"! *We originated what has become traditional "DeskTop Publishing"!* Today 68 Micro Journal is acknowledged as the "Grandfather" of "DeskTop Publishing" technology.

68 Micro Journal (ISSN 0194-5025) is published 12 times a year by Computer Publishing Inc. Second Class Postage paid at Hixson, TN. and additional entries. POSTMASTER: send address changes to 68 Micro Journal, POB 849, Hixson, TN 37343.

Items or Articles for Publication

Articles submitted for publication must include authors name, address, telephone number, date and a statement that the material is original and the property of the author. Articles submitted should be on diskette, OS-9, SK*DOS, FLEX, Macintosh or MS-DOS. All printed items should be dark type and satisfactory for photo-reproduction. No blue ink! No hand written articles - please! Diagrams o.k.

Please - do not format with spaces any text indents, charts, etc. (source listing o.k.). We will edit in all formatting. Text should fall flush left and use a carriage return only to indicate a paragraph end. Please write for free authors guide.

Letters & Advertising Copy

Letters to the Editor should be the original copy, signed! Letters of grip as well as praise are acceptable. *We reserve the right to reject any letter or advertising material, for any reason we deem advisable.* Advertising Rates: Commercial please contact 68 Micro Journal Advertising Department. Classified advertising must be non-commercial. Minimum of $15.50 for first 15 words. Add $.60 per word thereafter. No classifieds accepted by telephone.

Contents

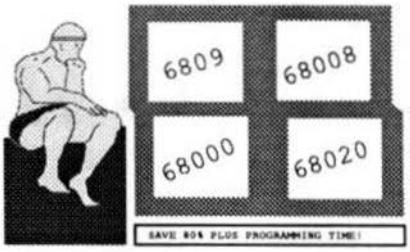
Features

Columns & Contributions

68 MICRO JOURNAL

CLOSE OUT SPECIAL

SCULPTOR



From the world's oldest & largest OS-9 software house!

CUTS PROGRAMMING TIME UP TO 80%

6809/68000-68030 Save 90%

SCULPTOR-a 4GL - Only from S.E. Media at these prices. OS-9 levels one and two (three GIMIX) 6809, all 68XXX OS-9 standard systems. Regular SCULPTOR versions 1.4:6. One of if not the most efficient and easy to develope DBMS type systems running under OS-9. A system of flexible keyed file access that allows extremely fast record and data retrieval, insertion and deletion or other programmed modifications. Access by key or in ascending order, very fast. The system provides automatic menu generation, compilation and report generation. Practically unlimited custom input format and report formatting. A rich set of maintenance and repair utilities. An extremely efficient development environment that cuts most programming approximately 80% in development and debugging! Portable, at source level, to MS-DOS, UNIX and many other languages and systems.

Standard Version: 1.14:6

6809 - $1295.00
68000 $1295.00
68020 $1990.00

Due to a "Special One Time" Purchase, We Are Making This Savings Offer. Quantities Limited!

Once this supply is gone the price goes back up!

System OS-9: 6809/68000-68030

• Regular ~~$1295.00~~

ONLY **$99.95**

S.E. MEDIA

POB 849 5900 CASSANDRA SMITH ROAD
HIXSON, TN 37343
TELEPHONE(615) 842-4601 FAX (615)842-7990

$AVE - WHILE SUPPLIES LAST!

OS-9000

REAL-TIME OPERATING SYSTEM AND DEVELOPMENT TOOLS

Technical Overview

OS-9000 is a modular real-time operating system and development environment that runs on the Motorola and Intel families of CISC and RISC microprocessors. Featuring a ROMable real-time kernel, file managers, utility functions and a variety of I/O and networking options, OS-9000 offers a full suite of development tools. These include a UNIX-like "shell" user interface, editors, compilers, source-level and system-state debuggers, plus C cross development environments for UNIX and PC-DOS machines.

Providing a superset of Microware's current OS-9 Real-Time Operating System, OS-9000 has been rewritten in C (95% of the kernel, 100% of the development tools) to maximize portability. Initially, OS-9000 will be available for the 68020, 68030, 68040 and 80386 micro-processors. By the first quarter of 1990, Microware plans to add support for Intel's 80486, as well as a variety of RISC processors (beginning with Motorola's 88000). Also planned for 1990 will be support for ISDN and a variety of loosely- and tightly-coupled VMEbus multi-processor architectures.

OS-9000 will be the first real-time operating system to support resident, as well as UNIX and PC-DOS cross development for both 680X0 and Intel 80386 microprocessors. The resident development capability enables designers to edit, compile and debug their code directly on the 80386 or 680X0-based target hardware. Cross development support, on the other hand, enables designers to develope their real-time code on a variety of UNIX and PC-DOS host, cross compile, and download their code to an 80386 or 680X0-based target.

To supplement the basic operating system and development environment, Microware will also offer a broad spectrum of LAN and backplane based communications options. It will also support the RAVE (Real-Time Audio/Video Environment) multi-media user interface. Combining audio, video, graphics and a customizeable menuing system in the same user interface, RAVE enables designers to configure realistic man-machine interfaces for real-time industrial and process control systems.

Scalable Architecture

One of OS-9000's most powerful features is its modular, scalable architecture. In OS-9000, all functional components, including the kernel, file managers, I/O drivers and development tools, are implemented as independent modules. By combining these modules, designers can realize a variety of OS-9000 configurations - from a lean, ROMable stand-alone kernel, to a full blown multiuser development system.

In a typical scenario, development is handled using a full featured configuration. Once the real-time code has been debugged, the development and I/O modules are stripped away and the kernel is deployed with the application code in the target system. All of OS-9000's modules are ROMable.

Recently, a number of other real-time kernels vendors have jumped on the "modularity" bandwagon. Typically, however, these kernels lack flexibility. Once designers have selected a configuration and generated the system, if they want to change the configuration and add or delete a module, they must recompile and relink the entire system. Because OS-9000 modules are position independent (modules are referred to name rather than by absolute address), they may be dynamically added or deleted from the system while the system is running without requiring recompilation or relinking.

A Compact, high-speed kernel

Because OS-9000 (and its predecessor, OS-9) has built a reputation on comprehensive development support, it's kernel-level capabilities are sometimes overlooked. In point of fact, OS-9000 includes one of the most robust kernels on the market, providing features such as:

- Mixed priority-based, pre-emptive real-time and time-shared scheduling modes.
- Task synchronization and process management constructs such as pipes, events, semaphores, and signals.
- 18-microsecond worst-case interrupt latency.
- Hierarchical tree structured directories.
- Written using re-entrant code (code that doesn't modify itself), thereby enabling the sharing of modules among processes and a reduction in memory requirements.
- Memory protection through support for user and system state process permissions.
- Enhanced memory protection by taking advantage of standard hardware MMU's (Though OS-9000 doesn't

require an MMU to run efficiently).

- o Standard C library interface for I/O operations (read and write calls) improves I/O performance and boosts application portability.

One of the most common misconceptions about OS-9 is that it provides only time-sliced task scheduling. Ironically, scheduling flexibility is one of OS-9000's most unique features. To maximize development efficiency, OS-9000 does offer a time-sliced multitasking scheduling mode. However, once designers are ready to deploy their application in a target system, they can also take advantage of OS-9000's real-time priority-based pre-emptive scheduling mode.

There are a number of stand-alone kernels that can match or exceed OS-9000's interrupt latency and context switch performance. Typically, however, they do so by reducing the flexibility of their systems calls and omitting system-level functionality such as memory protection. Context switch speed, for example, may be significantly increased by minimizing the context that the kernel must save. Interrupt latency, similarly, may be improved by off loading a variety of functions (such as resolving priorities for interrupts that share the same vector and informing the kernel that an interrupt handler has been entered) normally handled by the kernel to interrupt service routines.

The problem with stripping kernel functionality is that it complicates software development and adds overhead to application programs and interrupt handlers. In the long run, software developers must recreate and embed kernel-level functionality in their application, thereby degrading the system-level performance of their application.

Microware's approach to kernel design was to evaluate a broad class of real-time applications and define OS-9000's kernel-level functionality in terms of common application requirements. The end result is a robust kernel that not only simplifies application development, but boosts system-level performance.

Microware Systems Corp.
1900 N.W. 114th Street
Des Moines, Iowa 50322
Phone: 515/224-1929

Development Support

While kernel vendors continue to scrap over simplistic performance benchmarks, mounting time-to-market pressures are causing system designers to place a heavier emphasis on I/O capabilities and development support. It is in this regard that OS-9000 enjoys its chief advantage over its competitors. Development support for OS-9000 will include:

1. FORTRAN, C, and Basic optimizing compilers and interpreters, all of which generate compact, position independent, re-entrant, ROMable code.
2. User interface based on UNIX-like, "shell" style command interpreter.
3. Shell/Utility Set with over 72 commands for setting the "user" environment using functions such redirecting I/O, managing time, editing text, and monitoring processes.
4. Source-level debugger (for use with C compiler) that includes a C expression interpreter, on-line help, and support for breakpoints, single step, sub line skipping, variable manipulation, and monitoring of stack frame data.
5. Integrated symbolic debugger for the C compiler.
6. System state debugger.
7. Editors.

Unlike most real-time OS vendors, who integrate software components from a number of vendors, Microware designs all of its development tools and languages. As a result, we are able to provide a higher level of support for our products.

Resident, UNIX, and PC-DOS Cross Development

To increase development flexibility, Microware makes its development tools available on resident, as well as UNIX and PC-DOS hosts. Unlike most real-time kernels, which require a remote host to handle their development, OS-9000 supports a full suite of development tools on the target system (such as an MVME147 CPU board or a standard 386/PC, etc.).

This capability not only reduces the cost of the development hardware, since designers don't have to purchase a separate development system, it simplifies field maintenance and upgrades. Once the system is deployed in the field, updates and modifications can be performed locally without relying on the host. Even in a diskless environment, the entire OS-9000 operating system, together with a C compiler and debugger can be ROMed, thus allowing local modifications and recompilation.

UNIX Cross Development.

In addition to its resident development capability, OS-9000 provides a number of remote development options. For designers who prefer to handle their development on a familiar host such as a UNIX workstation or IBM PC, OS-9000 offers the UniBridge and PCBridge C cross development environments.

UniBridge is a C cross development package that links UNIX development hosts with real-time target systems running OS-9000. Combining C Cross compilers that run under UNIX with a source level debugger that runs under OS-9000, UniBridge links UNIX and OS-9000 via Ethernet using the TCP/IP protocol and BSD 4.2 sockets interface.

With UniBridge, designers develop their application program under UNIX on a host such as the SUN or VAX using the OS-9000 cross C compiler. The C compiler outputs executable OS-9000 memory modules that can be downloaded to the target OS-9000 system via Ethernet, and then debugged at the source or symbolic level

under OS-9000 on the target system. Support for Remote Login via Telnet and file transfer via FTP (File Transfer Protocol) enables users to access the entire OS-9000 operating system and debug their application from the UNIX host.

UniBridge is available for the Digital Equipment Corporation VAX, SUN workstations, Hewlett Packard Series 300 workstations and Apollo Domain systems.

PC Cross Development

While UniBridge supports UNIX cross development, PCBridge provides a comparable C cross development environment for PC-DOS-based machines. As with UniBridge, programmers develop and debug their application on a PC XT/AT (or compatible) using the OS-9000 development tools and a C cross compiler. It should be noted that OS-9000 can run resident on a 386/PC as well.

The PCBridge user interface appears as a pop up menu. Users select menu options (such as edit, compile, transmit, etc.) with either the keyboard or a mouse. Once designers have completed their application, they download it to the OS-9000 target system using a high-speed serial link.

Networking Support

To support the design of distributed architectures, OS-9000 will also support a wide range of networking options. OS-9000's Internet Support Package (OS-9000/ISP), for example, enables a host CPU board with integrated LAN hardware (Such as Motorola's MVME147, which includes the LANCE Ethernet chip set) to run the TCP/IP protocol without requiring an external communications controller board. When equipped with this ROMable software, any OS-9000 based 680X0 host can communicate with any other system (such as a UNIX, VMS, or DOS based system) running the TCP/IP protocol.

Initially, OS-9000/ISP will provide drivers for the LANCE Ethernet chip set. However, the software's device independence makes it easy to adapt to a variety of network interfaces, from Ethernet and ArcNet, to SLDC point-to-point serial links.

OS-9000/ISP conforms fully with the DARPA Transmission Control Protocol/Internet Protocol (TCP/IP). It also supports the DARPA specified File Transfer Protocol (FTP) and TELNET (virtual network terminal, also known as remote login). These protocols not only enable OS-9000 users to access other nodes on the network, but enable other nodes to access an OS-9000/ISP system as a server. An OS-9000/ISP BSD 4.3 UNIX socket-style communications interface provides the link between application programs and the TCP/IP protocol.

Complementing the Internet Support Package is an extension to OS-9000's Network File System that supports multiprocessing via RS232C cable, local area networks such as Arcnet (Ethernet drivers will be available soon), and the VMEbus system backplane. Known as OS-9000/NET, this software provides a logical extension to the OS-9000 I/O system, enabling users to access files and devices resident on remote systems as if they are resident on the user's own system.

OS-9000/NET works with any peripheral device, including other CPU's, printers, and mass storage devices. It supports all of the functions offered in the standard OS-9000 disk file manager, such as creating and deleting files and directories, and changing working directories.

Multimedia User Interface

To simplify the development of realistic man-machine interfaces for real-time industrial and process control systems, OS-9000 will also support RAVE (Real-Time Audio/Video Environment). Using RAVE, designers can combine high quality audio and video, computer generated graphics and customizeable menus in the same user interface. RAVE enables designers to build custom user interfaces and control panels without writing a single line of code. All of the indicators, controls and menus needed to configure an interface are available as library components that can be accessed and combined via RAVE's interactive, menu-driven editor.

If designers want to add components not already available in the library, they can build their own library functions. Alternatively, they can digitize the actual components using a video camera. In either case, the component can be modified and refined using RAVE's paintbox graphics package, and added to the remainder of the display.

Using RAVE's audio tools, designers can also capture, edit and play back audio segments. The audio may be input via any external source, such as a microphone or cassette, or loaded from disk. The completed audio can then be combined with the video image to complete the man/machine interface. The audio quality is limited only by the hardware.

Microware offices are located in Des Moines, IA, Santa Clara, CA, Southhampton, UK, Marseille, France and Tokyo, Japan, with field representatives worldwide.

For more information, please contact Microware Systems Corp., 1900 N.W. 114th St. Des Moines, Iowa 50322. Phone (515) 224-1929.

+++

C User Notes

The C Programmers Reference Source. Always Right On Target!

A Tutorial Series

By: Dr. E. M. 'Bud' Pass
1454 Lauta Lane N.W.
Conyers, GA 30207
404 483-1717/4570
Computer Systems Consultants

Introduction

This chapter presents references to books on algorithm design, a problem with a C program, and an example C program which replicates its input to its outputs.

ALGORITHM DESIGN BIBLIOGRAPHY

Thomas Kinsman (Eastman Kodak, Department 47, Rochester, NY) compiled a bibliography generated by a request for reference books on algorithms in general. Following are comments by Kinsman, then a summary of the bibliography and some quotes about the references.

It was amusing that there were several different combinations of titles and authors were given for "THE Dragon Book". The rumors are not clear as to whether the authors (Aho, Hopcroft, Sethi, and Ullman) are married to each other, or are simply joined at the waist from birth. You can all speculate about that among yourselves. Certainly they have made an important contribution to our field.

People generally mentioned Knuth. However, Knuth was frequently mentioned as a lower priority choice when priorities were mentioned. Many expressed difficulty with the way algorithms were given in Knuth. Others suggested that Knuth's method of presentation leads to a better understanding of the method of implementation.

Sedgewich was a popular first choice. However, no one mentioned that there is now a second edition of this book.

This was not a scientific survey. Votes were cast for a reference by mentioning it in a positive way. In the case where prioritized lists were received, no weights were assigned. Titles, Authors, etc... were taken on good faith. The existence of references was not checked.

Some people just responded by mentioning authors. Referring to a book as "Smith, and Jones" is not always helpful, especially if Smith and Jones are really married to each other and have jointly written several books. I did my best to figure out which book was intended, without killing myself in the process.

Some respondents just mentioned titles. Some books are so closely related, i.e. the "Numerical Recipies in..." series, that they were grouped together.

At any rate, this is just an information source for your consideration. Neither I, nor my company, recommend any of them. The following sources of information are some you may wish to know about.

Source: "Algorithms", by Robert Sedgewick
Votes: 19
Quotes: "...general purpose..."

Source: "The Art of Computer Programming",
by Donald E. Knuth
Volume 1: Fundamental Algorithms

Volume 2: Seminumerical Algorithms
Volume 3: Sorting & Searching
Votes: 39
Quotes: "...the last place I go..."
"...No question. Knuth..."
Source: "Intro. to Data Structures & Algorithms",
by Aho, Hopcroft & Ullman
Votes: 9
Quotes: "It's a great little book of basic algorithms
& data structures. Nothing too outrageous.
My copy is almost worn out."

Source: Data Structures + Algorithms = Programs,
by Nicholas Wirth
Votes: 6
Quotes: "...slightly easier to read, & includes
quite a bit of Modula-2 code..."
"...A notorious book...Disgusting..."

Source: Design & Analysis of Computer Algorithms,
by Aho, Hopcroft & Ullman
Votes: 6

Source: The Theory of Parsing, Translation & Compiling, by Aho & Ullman
Votes: 2

Source: Compiler Design,
by Hopcroft, Ulmman & Sethi
Quotes: These are the "dragon" books. <DRAGON>

Source: Principles of Compiler Design,
by Aho, Hopcroft & Ullman
Votes: 4
Quotes: The "dragon" book. <DRAGON>

Source: Compiler Design,
by Aho, Hopcroft & Ullman
Votes: 3
Quotes: This is the second Dragon book. <DRAGON>

Source: Intro. to Formal Languages, Automata Theory, & Computation, by Hopcroft & Ullman

Source: Handbook of Algorithms & Data Structures,
by G.H. Gonnet
Pub: Addison-Wesley, 1984
Votes: 4
Quotes: "I often look at [this book]. It is useful in itself
(with code in Pascal and/or C) but also full of references— to 23 textbooks & 683 papers to be exact."
Source: Writing Efficient Programs, by Bentley

Source: Programming Pearls, by Bentley

Source: Numerical Recipes in C, the art of scientific computing, Numerical Recipes in FORTRAN,
the art of scientific computing, Numerical Recipes,
the art of scientific computing,
by W.T. Vetterling, S.A. Teukolsky,
W.H. Press, B.P. Flannery
Pub: Cambridge, Cambridge University Press, 1988.
Votes: 8
Quotes: "...It doesn't matter which language, ..."
"...for numerical stuff..."

Source: Structure & Interpretation of Computer Programs, by Abelson & Sussman
Votes: 3
Quotes: "...for numerical stuff..."
by: Bender & Orszag
Quotes: "...for numerical stuff..."

Source: Artificial Intelligence Programming, by Charniac, Reisbeck, Mcdermott & Meehan
Quotes: "...It's not the most fundamental book, but I've found [it] to be the book that establishes the leap from 'Let's Learn Lisp!' to Lisp in the Real World - filled with all sorts of ideas, permanently beside my terminal."

Source: Recursive Techniques in Programming,
by D.W. Barron
Pub: New York, 1968: American

Source: Fundamentals of Data Structures,
Fundamentals of Data Structures in Pascal,
by Horowitz & Sahni
Votes: 5
Quotes: "...my personal favorite..."

Source: Principles of Data Structures & Algorithms,
by Horowitz & Sahni
Votes: 2
Quotes: "...for general data structures..."

Source: Fundamentals of Computer Algorithms, by Horowitz & Sahni

Source: Computers & Intractability: A Guide to the Theory of NP-Completeness, by Michael R. Garey & David S. Johnson.
Votes: 2
Quotes: "...Perhaps not an algorithm catalog in the strict sense, but I find it useful in problem solving..."

Source: Heuristics, by Pearl
Quotes: "...if the problem is NP complete..."

Source: How to Solve It by Computer, by Dromey R.G
Pub: Prentice/Hall International Series in Computer Science

Source: Combinatorial Optimization: Algorithms & Complexity, by Christos H. Papadimitriou & Kenneth Steiglitz
Quotes: "fairly new but it is full of interesting stuff. I looked at several similar ones & bought this."

Source: Computer & Job-Shop Scheduling Theory by E. G. Coffman, Jr. (Ed.)
Quotes: "A collection by several recognized people."

Source: "Factorization Methods For Discrete Sequential Estimation"
Quotes: "useful for estimation algorithms."

Source: Data Structures & Network Algorithms, by Tarjan
Quotes: "...for network problems..."

Source: Graphs & Network Algorithms, by Tarjan
Quotes: "...for combinatorial algorithms & recursive structures..."

Source: Priciples of Database & KnowledgeBase Systems, by Ullman
Votes: 2

Source: Implementations of PROLOG, by Campbell

Source: The Computer Modelling of Mathematical Reasoning, by Bundy

Source: Anatomy of LISP, by Allen
Source: Natural Language Understanding, by Allen

Source: Data Structures, by Reingold & Hansen

Source: Data Structures, by Standish
Votes: 2
Quotes: "...understandable level with good Knuth-style specifications. Quite complete also..."

Source: The Unix Programming Environment, by Kernighan & Pike

Source: Software Tools, by Kernighan

Source: Matrix Computations, by Holub & Van Loan
Quotes: "...for numerical analysis..."

Source: Artificial Intelligence, by Winston
Quotes: "...for AI work, either of the books by Winston..."

Source: LispCraft, by Wilensky

Source: Computer Algorithms, by Sara Baase

Source: Lisp, by Winston

Source: Prolog programming for AI, by Bratko

Source: Fundamentals of Interactive Computer Graphics, by Foley & Van Dam
Votes: 2
Quotes: "...for graphics stuff..."

C PROBLEM

Following is a program written by Mike Segal. The comment describes a problem with it. Can you determine the problem without reading ahead to the solution below? For those with systems providing lint, the problem is much more obvious.

```
/
**********************************************************/
/*                                                        */
/* Generic adder / multiplier          */
/*                                                        */
/
**********************************************************/
/*
This program is to take two numbers and add them
together.

For test data I have the two numbers to be 8 and 40320.

The solution should be 823040 but, using the cc com-
piler, the solution was 823000.

The 4 was never output.

For debugging purposes, I had inserted a printf to
display the value for each digit (now commented out).

When recompiled, the program gave me the correct
solution.

BUT when I removed the print statement, it gave me the
incorrect solution.

Mike Segal
*/
#include <stdio.h>
char *calloc();
typedef struct
{

int len;
char *cray;
}

NUM;

main()
{
NUM gadd();
NUM a, b, c;
int i, j, k;

a.len = 5;
a.cray = (char *)calloc(a.len * sizeof(char));
if (a.cray == NULL)
(void)printf("A Null!\n");
b.len = 1;
b.cray = (char *)calloc(b.len * sizeof(char));
if (b.cray == NULL)
(void)printf("B Null !\n");
c.len = 1;
c.cray = (char *)calloc(c.len * sizeof(char));
if (c.cray == NULL)

(void)printf("C Null !\n");
*(b.cray) = 8;
*(a.cray) = 0;
*(a.cray + 1) = 2;
*(a.cray + 2) = 3;
*(a.cray + 3) = 0;
*(a.cray + 4) = 4;
c = gadd(b, a);
(void)printf("The solution is \n");
for (i = 0; i < c.len; i++)

(void)printf("%d", *(c.cray + i));
(void)printf("\n");
}

NUM gadd(x, y)
NUM x, y;
{

NUM sum;
int i, j, k;
char carry;
char *spt;
/* Initalize sum */
if (x.len >= y.len)
sum.len = x.len + 1;
else

sum.len = y.len + 1;
sum.cray = (char *)calloc(sum.len * sizeof(char));
spt = sum.cray;
/* Add loop */
j = 0;
carry = 0;
while ((j < x.len) || (j < y.len))
{
*spt = carry;
if (j < x.len)
*spt += *(x.cray + j);
if (j < y.len)
*spt += *(y.cray + j);
if (*spt > 9)
carry = *spt / 10;
else

carry = 0;

*spt %= 10;
/* (void)printf("\n"); */
spt++;
j++;

} /* End of while */
if (carry)

*spt = carry;
```

```
return(sum);

} /* End of Generic Adder */
```

The problem is with the calls to calloc. Although malloc requires one argument (the number of bytes requested), calloc requires two arguments (the number of elements and the size of each element). Replacing the multiply symbol with a comma in each of the calls to calloc corrects the problem.

This problem illustrates only one of the inconsistencies in the standard C language library. Programmers must be very careful and use whatever tools are available, such as lint, program beautifiers, etc. to prevent errors such as this one from creeping into their programs. Since functions such as malloc and calloc provide such similar capabilities, it is easy to forget that they have different calling sequences.

EXAMPLE C PROGRAM

Following is this month's example C program; it clones its standard input to its standard output and to all files listed on the command line, similar to the tee command in several popular operating systems.

Note the use of the fread and fwrite functions, rather than the getc and putc or similar functions, for performance reasons. Characters are processed in blocks, rather than individually, on each function call, drastically reducing the average overhead associated with each character.

```
#include <stdio.h>

#ifndef BUFSIZ
#define BUFSIZ 512
#endif

#ifndef MAXFILE
#define MAXFILE 20
#endif

FILE *outfil[MAXFILE];
char in[BUFSIZ];
int k;
int n;
int t;
int w;

main(argc, argv)
int argc;
char **argv;
{

outfil[0] = stdout;
for (k = n = 1; (k < argc); ++k)
{

if (outfil[n] = fopen(argv[k], "w"))

++n;

else
{

(void)fprintf(stderr,

"%s: Cannot open %s\007\n",
argv[0], argv[k]);

}

}
while (w = fread(in, 1, BUFSIZ, stdin))
{

for (k = 0; (k < n);

(void)fwrite(in, 1, w, outfil[k++]));

}
for (k = 1; (k < n); (void)fclose(outfil[k++]));
(void)exit(0);

}
```

RAVE

A Multi-Media User Interface Package for OS-9/68000

Tim Harris and Lee Glenn
Microware Systems Corporation
Des Moines, Iowa

Abstract

In recent years, there has been an increase in user interface research and user interface packages for computing systems. As these computing systems move more toward the consumer and factory floor, a new type of interface is required that is more intuitive and natural for the user. The Real-Time Audio/Visual Environment (RAVE) package for the OS-9/68000 operating system is a multi-media user interface package designed to meet these goals. With RAVE, developers can quickly create control applications which utilize natural images of real world objects and incorporate audio as well as visual feedback.

Introduction

In the past several decades, there has been an increased emphasis on the interface between computers and their operators. In the early days of computing, this interface was physical; the user interacted with the machine via switches. Next came impersonal interfaces; the user fed his data or programs into the machine with punch cards or paper tape. The first revolution in the human-computer interface was the video display terminal. For the first time, an individual user could sit down at a terminal and interact directly with the computer by typing in commands at the keyboard and seeing the results on the display.

In the mid 1970's, XEROX Palo Alto Research Center was developing a graphics based user interface in which the user could interact with objects on the screen in one of several windows. Each window was analogous to a piece of paper on a desktop. Along with multiple sheets of paper, the user also had several desk "tools" which could be used.

This window/graphics technology was introduced to the public by Apple with the introduction of the Macintosh in the early 1980's. The Macintosh was designed to be the first computer that could be intuitively used by anyone. It offered a consistent graphics interface which was easy to use and learn.

Today, a graphics/window interface is almost expected as standard equipment on home and business computer systems. Among the standard offerings are Microsoft Windows, the OS/2 Presentation Manager, the Amiga Workbench, GEM and X Windows for UNIX based systems.

These new interfaces have had a great impact on the efficiency of computing for professional users. These systems have also made it easier for non-computer professionals to learn how to use a computer. A person no longer has to learn dozens of commands with hundreds of options. Instead, they learn the meaning of specific icons and how to manipulate objects in the specific environment.

These interfaces, though much more intuitive compared to the previous command line interfaces, are still hard to use for some people. This is especially true for non-computer oriented, illiterate or pre-literate people. They are simply overwhelmed by interaction with windows and graphic icons which stand for real world objects, but don't look or act exactly like the real world object. Development time for applications under these interfaces may also be extensive. Programs which make use of many menus and objects tend to become large and complex. In many cases, there is much to learn about the specific interface before an application can be written.

As more powerful hardware, coupled with high quality video and audio, becomes the standard, a user interface is required that virtually anyone can use. It must be an interface where things look, act and sound so much like real objects that the user needs very **little or no training**.

Along with being extremely user-friendly, the interface must also be easy to use from the programmer's viewpoint. Objects on the display would be best handled separately from all other objects. Consequently, a change in one would not require re-writing code throughout the program. In fact, it would be best if a programmer did not have to write the code for these objects at all, but could merely define objects and have the corresponding code automatically generated.

RAVE (Real-time Audio Visual Environment) from Microware Systems Corporation is such an interface. RAVE is a true multi-media user interface designed specifically to build objects which look, sound and act like their real world counterparts. Not only does RAVE give the application a natural looking interface but it also gives the programmer or application designer a way to interactively build objects and have source code generated for them so that little time need be spent in actually writing code for the application.

Microware Systems Corp.
1900 N.W. 114th Street
Des Moines, Iowa 50322
Phone: 515/224-1929

Western Regional Office
4401 Great America Parkway
Santa Clara, California 95054
Phone: 408/980-0201

Microware Japan Ltd.
41-19 Honcho 4-Chome
Funabashi City
Chiba 273, Japan
Phone: 0474 (22) 1747

The Development of Rave

Rave and the CD-I Environment

RAVE was developed from Compact Disc Interactive (CD-I) technology. In 1986, N.V. Philips of Holland and Sony of Japan joined together to create a new standard for optical media. The first true multi-media disc would handle computer data, natural image data, high quality audio data and have the ability to play all of these back in a synchronized manner. This new disc format was called CD-I and the standard became known as the Green Book.

When the CD-I specification was first being developed, Philips and Sony came to Microware Systems Corporation for the operating system to drive these new players. This version of the OS-9/68000 operating system is known as CD-RTOS: the Compact Disc Real-Time Operating System. Microware became closely involved with the CD-I Green Book specification.

At the same time, Microware saw the need for a new user interface for these systems. This user interface needed to be usable by a consumer, not a computer user. It had to be so intuitive that ordinary consumers and even children could use it with virtually no training. It also had to take advantage of the high quality audio and natural image capabilities of the CD-I players.

In 1987, Microware began a joint development effort with Thomson ISD to develop this new interface. It became known as CD-RTOS/RAVE. The philosophy behind RAVE development was to create an event driven user interface which would allow developers to use natural images and high quality audio in the interactive objects on the display. No visual restrictions for objects were to be made. For example, a button did not have to be a rectangle with text in it. Instead, it could be a natural image of a button captured with a video camera. Audio feedback did not have to be a warning "beep;" it could be a real voice telling the user what is wrong.

Rave and the OS-9 Environment

After the initial development of CD-RTOS/RAVE was completed, it was decided that RAVE would be useful not only in CD-I systems, but also in the process control environment. Current machine front panels could be replaced by computer terminals that display the same front panel switches, lights and meters. The advantage of a software based front panel is that one

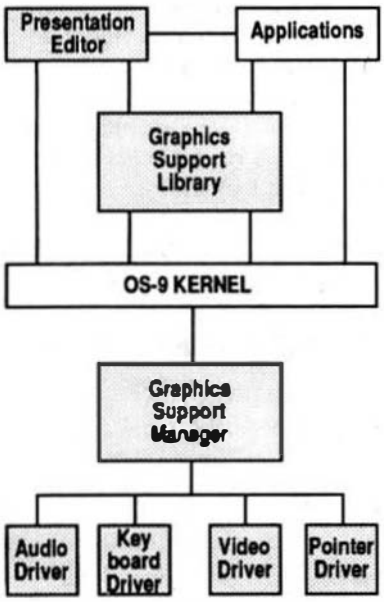
display can be used to "run" several different pieces of hardware or the layout of the objects on the display could be rearranged or altered for better efficiency. This would promote a user interface that is easy to learn and use for factory floor applications.

Consequently, a more portable general purpose version of RAVE was created for any OS-9/68000 based system. New objects were added to the user interface to make it more applicable to the process control environment. A new tool, the OS-9/RAVE Presentation Editor, was also added to make development of RAVE applications easier and faster. With the new RAVE development tools, a programmer no longer has to do all of the work in writing and designing an application. The programmer writes a minimal amount of code and an artist or designer can create and lay out the objects to be used on the actual display.

THE RAVE ENVIRONMENT

OS-9/68000 is a modular operating system allowing various types of memory modules to reside in memory to build up the software base of the system. RAVE takes advantage of this modular nature and is built up of several layers of software from the system level up to the actual application building software at the user level:

- o System Level Software:
  The Graphics File Manager and Drivers
- o Library Level Software:
  The Graphics Support Library
- o Application Design Software:
  The Presentation Editor



RAVE SYSTEM LEVEL SOFTWARE

The Graphics File Manager (GFM)

The highest level of the I/O structure under OS-9 is the file manager module. A file manager is responsible for handling the raw data for a class of similar devices. For example, the Random Block File Manager handles all random access block structured devices such as disk drives. For the RAVE environment, the Graphics File Manager (GFM) handles all I/O for the multimedia devices. These devices consist of video and audio output devices and keyboard and pointer input devices.

GFM has many functions. It directly handles opening and closing I/O paths to these devices. It dispatches calls to the appropriate devices to handle graphics drawing, character input and output, pointer input, and audio output. On top of these standard file manager duties, GFM also supplies the RAVE environment with the support needed for multi-tasking applications. This is accomplishes by supplying the application programmer with a construct known as a Screen. Screens may be opened for output and can be switched under application or user control. Screens are the base objects an application must have open to display objects on a video device.

To handle user input for an interactive, event driven environment, GFM supplies a facility for generating application messages relating to pointer activity on a Screen and global changes in the environment. Messages are generated by a system state process that obtains information from the pointing device and builds messages. The messages provide the user with specific information, including:

- o the coordinates of the pointer
- o the state of the buttons
- o the time of the message
- o whether the pointer is in any specially defined area of the Screen

Messages are also generated by GFM to notify an application that its Screen is activated or deactivated and that the user has changed the system parameters.

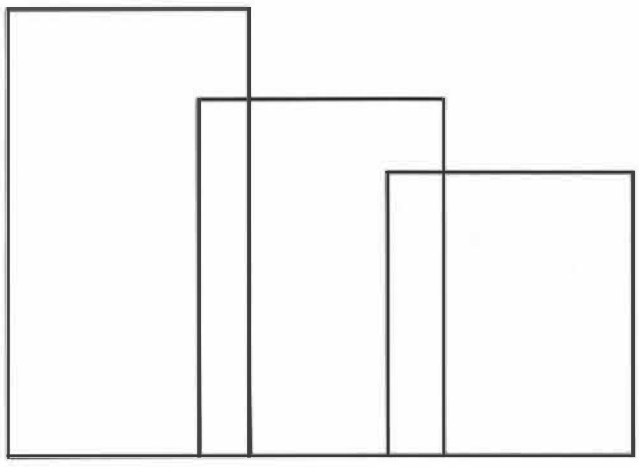
GFM also manages certain "hot spots" on the Screen. These hot spots are areas in which the application may be especially concerned about a certain type of input activity. These hot spots are known as Action Regions in RAVE terminology.

An Action Region is an arbitrary region of a Screen which can be used to filter out messages for the application. For example, there may be a rectangular portion of the Screen where a button will be placed. If the application wishes to know specifically when the pointer button is depressed within this area, it can define this area as an Action Region and ask that only "button down" messages be generated for this specific Region.

Action Regions have various characteristics associated with them. They have a size and shape, a cursor shape and color used for the graphics cursor when it enters the Region, and a set of message masks used to filter messages for the Region. There is a basic message filter, a message "grabbing" mask and a "terminate grab" mask for the Region. When a message in an Action Region matches the "grabbing" mask, all new messages generated are "grabbed" by that Action Region until the application terminates grabbing directly or until a message that matches the "terminate grab" mask is received.

Action Regions are arranged in a hierarchical tree structure. Therefore, an Action Region may have a single parent and multiple children. All child Action Regions inherit the base characteristics of the parent but can be modified.

GFM has a full set of text functions that allows multiple, application defined fonts and various character encoding methods. Because RAVE is designed for an international market, text encoding is not limited to standard 8-bit ASCII codes. GFM also supports 7/15 bit and 16 bit codes to support non-European character sets such as JIS Kanji. In 8 bit character mode, GFM contains the full functionality of OS-9's standard terminal file manager, the Sequential Character File Manager (SCF). This allows a GFM device to be used in terminal emulation mode in order to run standard OS-9 terminal based utilities such as the OS-9 Shell.



The GFM Drivers

Directly under GFM is a set of device driver modules which handle the low level I/O for a specific type of hardware. There may be one or more physical devices controlled by each driver. For example, a system could have two video cards named "/v0" and "/v1" which are both managed by the same video driver. The characteristics of the physical device are kept in a small module called a device descriptor which defines the address, interrupt vector, interrupt level and other parameters for the specific hardware.

Unlike most OS-9 file managers which communicate with only one type of device driver, GFM communicates with four types of drivers: a video driver, an audio driver, a keyboard driver and a pointer driver. GFM ties all of these devices together into a logical "multi-media" or "workstation" device. The application program only has to open a path to the main device in the chain (usually the video device) and all other devices are attached and opened for the application by the GFM.

The Video Driver. The most complex of the GFM drivers is the video driver. This device driver is responsible for handling all output to the actual video hardware. The video driver handles all of the graphics drawing, text output and video memory allocation.

Architecture of the Video Driver. The RAVE video driver was designed to be easily ported to various types of hardware from simple bitmapped hardware to more sophisticated graphics co-processors. The video driver is built from three distinct code segments: the common code, the mode dependent code, and the hardware dependent code.

The common code handles the high level constructs of the video driver and does all the scan conversion for graphics drawing. This is by far the largest portion of the driver and comprises about 80% of the code. This part of the code usually does not change from driver to driver.

The mode dependent code is the actual blitter code used to combine and move pixels for each general type of video. There is code for 8-bit Color Look Up Table (CLUT) hardware and 4-bit CLUT hardware. The mode code makes up about 10% of the driver code.

The last 10% of the driver code is made up of hardware dependent code. This is the low-level code which must be written for each new video card. The hardware dependent code allocates and deallocates video memory, displays Drawmaps, and sets/gets both pixels and CLUT values in the hardware.

Constructs of the Video Driver. The base construct of the video driver is the Drawmap. A Drawmap is a pixel map which contains the actual video data for a display. A Drawmap is allocated by the video driver and can be linked to a GFM Screen to be displayed on the video device. All graphics drawing functions take place on a specified Drawmap. Drawmaps can be virtually any size, but in order to be linked to a Screen and displayed, they must be the size of the actual video display.

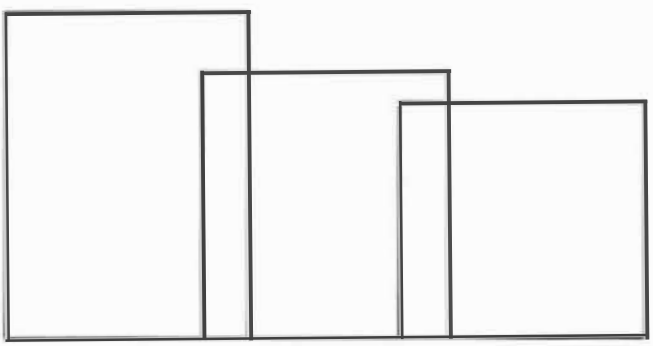
There are several parameters which may be specified to describe how pixels will be set in a draw-map:

- o Pattern or non-pattern drawing
- o Pixel mixing mode
- o Solid or Dashed line drawing
- o Pen width and height setting
- o Transparency in drawing
- o Outlined or Filled objects

The Audio Driver. The RAVE Audio Driver is used to handle allocation of audio Soundmaps and the playing and recording of Soundmaps. A Soundmap is a block of memory which contains sound data for an application to be played through the system's audio device. Depending on the hardware, the Audio Driver has entry points to allocate, delete, play and record Soundmaps.

The Pointer Driver. The RAVE Pointer Driver is responsible for reporting pointer activity to GFM's message handler or directly to the application. The pointer device can be a mouse, joystick, trackball, lightpen, graphics tablet or touch screen. The only requirements are that the device must generate x,y coordinates and must have one or more triggers that can be accessed.

The Keyboard Driver. The RAVE Keyboard Driver handles all input from a keyboard or keypad for the application. Keyboard drivers can be written for different types of keyboards; from a full terminal keyboard to a small keypad with just a few keys. The keyboard driver needs to be able to read the keyboard and return the character code of the key(s) which are depressed.



THE LIBRARY LEVEL SOFTWARE

On top of the RAVE System Level software is the RAVE Library Level software: the Graphics Support Library (GSL). GSL is supplied as a linkable C library or an OS-9 Trap Handler module which can be loaded into memory and shared by several applications running at the same time on one copy of the code. GSL contains routines to handle higher level objects in the RAVE environment. These objects are built from the low level Action Regions and Drawmaps and are used directly for user interaction. GSL objects are divided into four major classes: Requests, Controls, Indicators and Overlays.

Requests

A Request is an object used to get user input concerning a choice to be made among many alternatives. In common terms, Requests are used to generate menus for the application. GSL Requests can be either modal or modeless in nature.

A modal Request requires input directly before any further interaction can take place. Modal Requests will stop all other input on the display until a selection (or non-selection) is made. The item number of the selected item is returned directly to the application to let the program know what choice the user has made. An example of a modal Request is a pop-up or pull-down menu.

A modeless Request can remain active on the display for an extended period of time. The user can repeatedly make selections from it. A modeless Request interacts with the application using call-back functions. These functions are automatically executed when the user makes a selection from the Request. Each Request item can have a corresponding call-back function. When the user selects an item in the Request, that function is called directly. An example of a modeless Request is a standard menu bar.

GSL supports image based and text based Requests. A text based Request is made up of text strings for each item in the Request. It is similar in nature to most menu systems found on popular user interfaces. GSL also supports image based Requests which use different images to show the different states of each item in the Request. For example, when the user moves the pointer onto an item in the Request, the item may be changed by supplying a different image for that state. Items may be defined for the disabled, normal, pointer on item, and pointer depressed on item states. Audio Soundmaps can also be included for each item in the Request.

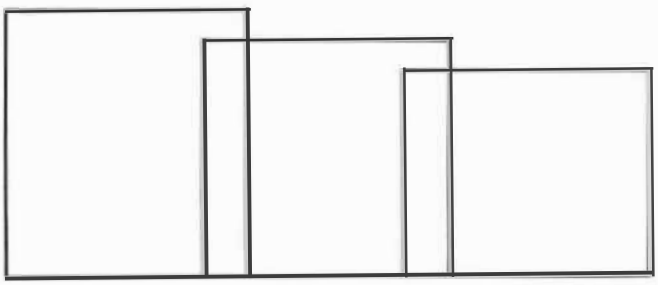
Along with the standard Request forms, GSL allows the application designer to define custom Requests by simply writing a new Request Definition Function which describes the Request.

Controls

A Control is an input object that is used to simulate real-world push buttons, toggle buttons and slide bar type devices. A Control is an image based object: the application designer supplies images for the background of a Control and the various states of the Control. A simple push button, for example, is defined by supplying a background image of the button and two thumb images for the button: one with the button up and one with the button depressed. When the user clicks the pointer on the button, the button will appear to be depressed just as a real button would. Toggle buttons are handled in a similar manner with the application designer supplying different images for the different values of a toggle button. A three position switch, for example, would be made up of a background image and three separate thumb images for each of the three switch positions. Soundmaps may also be provided to add sound, when the control is used.

Controls communicate input from the user to the application using call-back functions. When a Control is used, these functions will be directly called to initiate the action of the Control. For example, if an application has a slide bar control for audio volume, the call-back routine would be responsible for changing the volume based on the value of the slide bar Control. Whenever the user moves the slider, this function will be executed and the volume will be changed in real-time.

Controls are defined by both their appearance and their behavior. These two qualities are controlled by two functions known as the Control Definition Function and the Control Behavior Function. GSL supplies a standard image based Definition Function and two behavior functions for buttons and sliders. GSL allows the application designer to define custom Controls by simply writing a new Control Definition and/of Behavior Function which describes the Control.



Indicators

While Requests and Controls provide input objects for an application, Indicators provide output objects. Indicators are used to display a value to the user in one of several forms. The following standard Indicators are currently supplied with GSL:

- o Linear Meters
- o Strip Recorders
- o Level Indicators
- o LED Indicators
- o Readout Indicators

An application designer can easily write a custom Indicator Definition Function to define the characteristics of a new class of Indicator.

Indicators, like Controls, are image based objects. The application designer supplies a background image for each Indicator and specifies the colors to be used for any changing part of the Indicator. For example, to create linear meter, the application designer would supply an image of the meter and a color value for the needle of the meter to use for updating its value.

All Indicators have an application defined range associated with them which can contain sub-ranges for feedback. For example, a meter may have a total range of 0 - 100, with the range of 0 - 20 defined as dangerously low and the range from 80-100 defined as dangerously high. An Indicator could be created to monitor these ranges and automatically play a SoundMap and execute a call back function whenever the meter moves into or out of the low or high range.

Miscellaneous Functions

In addition to supporting the main objects of Requests, Controls and Indicators, the GSL has several other features. One important feature is Overlay Windows. An Overlay Window is a temporary window on the Screen which can be used to display error messages, dialog boxes or pop-up type menus. An Overlay Window can save the area which it obscures. The display can then be restored without any application re-painting when the Overlay Window is removed. GSL supports several pre-defined frames for Overlay windows which include an outlined frame, a double outline frame and a shadowed box frame. Any Request, Control or indicator may be placed on and used within an Overlay Window.

GSL also supports a Clipboard device for inter-or intra-process data exchange. Data in the Clipboard is stored in data modules in memory, but may also be written to or read from a novolatile storage device. Clipboard data may be ASCII text, pattern data, image data or audio data.

SL supports a set of internationalization routines to format monetary amounts and date strings for the application.

THE APPLICATION DESIGN LAYER

The OS-9/RAVE Presentation Editor was developed as an application designer's tool for implementing the features of RAVE in an application as easily and as quickly as possible. The Presentation Editor includes a rich set of video and audio manipulation tools as well as the facility for building GSL objects and generating C language source code for them. The Presentation Editor is an interactive graphics based application built on RAVE. It uses menus and dialog boxes for all its work: from showing images to defining the parameters for a Control.

Building a RAVE application consists of four steps:

1. Capturing or creating video and audio data that will be used by the application.

2. Building the required controls, requests, indicators and screens with the Presentation Editor.
3. Writing the application "front end" and call-back functions.

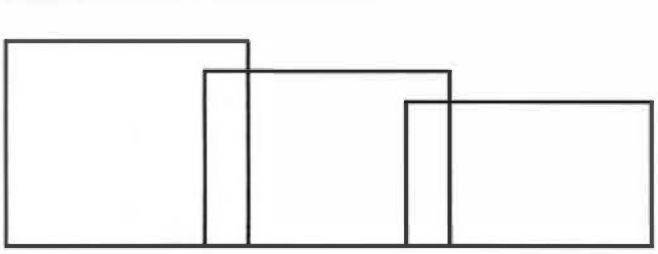
4. Compiling, linking and testing.

Video Tools

The Presentation Editor includes tools for the following:

- o grabbing live video (if the hardware permits)
- o saving and loading the video file
- o importing/exporting to other file formats
- o cutting and pasting portions of the image
- o resizing
- o adjusting the hue
- o complementing
- o converting to grayscale
- o smoothing
- o mirroring
- o rotating

A Paint Box facility is provided to allow drawing with lines, boxes, arcs, circles and ellipses. Text can also be added with the Paint Box tool.



Audio Tools

In the audio portion of the Presentation Editor, there are tools for capturing live audio (also hardware dependent), playing an audio file, cutting out a portion of the file, adding silence to a file, and merging two audio files together.

Code Generation Tools

The code generator can build and generate C source code for all GSL objects interactively with a keyboard, pointer and video display. When building an object, the designer is prompted for all necessary data. For example, the designer only has to fill in the blanks for items such as the name of the audio file to play when a Control is activated or click on check boxes to select the various indicator parameters. The designer is also prompted for the name of the call-back function to be executed when the object is activated. This call-back function and the small "front end" function are all the code that the designer will have to write.
For each object created, a C source file will be generated which includes the definition of that object and several routines that act directly upon the object to initialize, free, and update or interact with it depending upon the class of the object. Each object is a separate entity that can be modified without effecting any other object in the application.

After building the objects, the screen builder allows the designer to build the screen by adding these objects to a background image. The actual location of each object is specified by the designer. Objects can be added to the screen, moved or deleted until the final screen is designed.

When all source code has been generated, the designer must write the application specific code and an OS-9 "makefile". The application specific code consists of three major sections:

- o initialization code
- o open the video path and initialize the display
- o set up application required color registers
- o initialize the first Screen
- o a message dispatch loop
- o call-back functions for Controls and Requests

The bulk of the coding to be done by the application designer is in the call-back functions. These functions must carry out the actions for any Control or Request item with which the user is interacting. In many cases, the call-back function will simply update another object or cause a new
screen or overlay to be invoked.

After the coding is done, the designer can invoke the OS-9 MAKE utility from within the Presentation Editor to compile and link the application. Once the application is created it can be executed and any debugging or changes made to it. For example, if the designer decides a horizontal slide bar would be more appropriate than a vertical one, the object can be loaded into the Presentation Editor, modified and the application remade in a matter of minutes.

CONCLUSION

Today and in the future there will be a greater need for very intuitive and natural interfaces between computing systems and their users. RAVE offers a package that supplies the basic tools to create such interfaces now. RAVE was created out of the evolving CD-I consumer technology to provide a process control environment that is fully multi-tasking and has multimedia objects which can mimic real world objects in appearance, behavior and sound. This allows a natural, easy to use interface to be created; an interface that can be used without extensive training or computer experience.

With the RAVE development tools, a new approach to programming is available. A sophisticated interface does not require many man hours of programming with RAVE. Instead an application designer can spend time interactively designing objects and laying out screens with the user in mind. The programmer is responsible for only a small amount of actual hand-written code.

Finally, because each object can be modified independently, application designers can focus on creating an optimal interface without the extensive re-writing of code.

About the Authors:

Tim Harris is a Senior Software Engineer for Microware Systems Corporation. Tim was one of the original designers of the RAVE software for CD-RTOS and has been involved in the project since that time. He received a B.S. in computer science and psychology from Iowa State University in 1984.

Lee Glenn is a Software Engineer also with Microware Systems Corporation. Lee has been involved in graphics, CAD and image processing for a number of years and wrote the first CD-I application. Lee has degrees in computer science from Simpson College and mechanical engineering from the State University of New York.

OS9 USER NOTES

By: Peter Dibble
49 Stony Lonesome Road
Homeoye Falls. NY. 14472

The world was much different when OS-9 was designed. We thought of 16K of RAM about the way we think of 4M these days: a system with 32K of RAM was big. Five inch floppies held about 100K. and hard disks were out of the question.

Expensive RAM was a manageable problem when simple programs ran one at a time. but even back then it was plain that eight kilobytes of RAM was not going to make it in a serious multitasking system. Typical programs were getting bigger and the multitasking that was designed into OS-9 needed several programs in memory at the same time. ROMs were a way to cut RAM requirements and reduce (or eliminate) the effect of slow disk drives.

Mask programmed ROMs were inexpensive compared to RAM. A system that could run with most of its software in mass-produced ROM could get by with less RAM. and mass storage for programs would not be an issue. Motorola latched onto this idea and it became one of the assumptions used to design OS-9. This design goal had broad influence on OS-9. and traces of it survive to this day.

It would not be practical to produce mask programmed ROMs for each system configuration. Small runs of masked ROMs were (and still are) expensive. One solution would have been to create one standard ROM full of wonderfully flexible software then hope that everyone would be happy with the standard package. Instead. OS-9 treats ROMs as building blocks and the system finds ROMs and organizes the data they contain. Provided that the software is well designed. it's easy to assemble a collection of ROMs into a deeply specialized package. Even the OS-9 operating system is a set of components that fit together when the system is booted.

A basic problem with "software in silicon" is addressing. What if someone wanted to install a math package and a spreadsheet. but both of them needed to go in the $7800 ROM slot? Should ROM manufactures sell a version of the ROM for every possible address? OS-9 solves this problem with position independent code and data. A software chip will work at any address.

ROMed software posed plenty of other problems. for instance:

*How can the system find and incorporate ROM software during bootstrap?
*What if there is a bug in some ROMed software?
*How can software from disk be used to augment a ROMed system

For that matter. how can one ROM update software in another?

CRC

The bootstrap problem is enough to explain OS-9's modules and CRC codes. During the OS-9 bootstrap procedure. the kernel searches through memory for ROM and RAM. It identifies RAM by storing two different values into the same location and reading them back. ROM and addresses with no memory both fail the RAM test. The kernel takes special inter-

est in addresses that fail the RAM test because they might contain ROMed software.

OS-9 packages software in carefully marked "modules" so it can be distinguished from junk. A module starts with a sync code, then there is a module header with a parity check. The most rigorous test is a 24-bit CRC code stored at the end of the module. The sync code is easy to recognize, but it could show up anywhere. The kernel would find lots of false modules if the sync codewereall ithadto go on. After the kernel has found a sync code and a correct header, chances are good that it has a real module. The CRC code is a mathematical function of the rest of the data in the module. The OS-9 kernel checks the CRC code for each proposed module. If the calculated CRC matches the code stored where the end of the module should be, the kernel is satisfied that the memory contains a valid module and it lists the address of the module header in the module directory.

A module is a package of data (usually a program). Along with the distinguishing marks that help the kernel recognize it, each module module header contains codes that characterize that module: data like the module name, its type, the language code it uses, and its RAM requirements.

Plug a ROM with a graphics package into an OS-9 system, then boot the system. The graphics package will be in the module directory ready to run. Any programs in the ROM can be run by naming them in the OS-9 shell. Other module types, such as subroutine modules and device drivers, are known by name to other modules and dynamically linked.

Version Numbers

It used to be rather common to package the most common utility programs on one ROM. If all software were bug free and equipped with every useful feature, it would be fine to have everything cast in silicon, but what do you do when one of the utilities needs a one-byte patch? Get a whole new ROM? OS-9's answer is kernel support for version numbers. When the kernel has a choice between two modules with the same name it chooses the module with thehigher version number.

To patch a module in ROM you follow these steps:

* Save the module to disk
* Make the patch on disk
* Increment the version number of the module on disk
* Correct its CRC code.

When the revised module is loaded into memory it will supplant the module in ROM.

Dynamic Binding

Modules, including the OS-9 kernel, don't know where to find other modules. All they can know is the names that other modules will use, but that is enough. (Think of modules as analogous to files on a disk. A module's address is like the sector number where the file starts. It's not the kind of thing you want to hard-code into a program.) Given a module name, the F$Link system call will return an address that will reach that module. Complicated software often consists of several modules that find one another at run time. Since each module is found by name, the behavior of the system can be changed by substituting a different module with the same name.

This dynamic-binding feature is so commonly used that it is taken for granted. One example is the handling of the 68881/2 math coprocessor under OS-9/68000. A program may choose to have complex mathematical functions executed by a standard trap handler named "math." The default math module does all its calculations using the 68000 instruction set, but there is an alternate math module that uses the 68881 coprocessor. Programs will use whichever version of math is in memory, so software can be upgraded to use the coprocessor quite painlessly. The math module that uses the 68881 is smaller and faster than the math module that does IEEE floating point in software but that is the only visible difference.

I frequently benchmark software with the software version of the math module. I copy math.software into math in my CMDS directory. Next time I boot the system it comes up using software floating point. (I could do this change without rebooting, but this way is easy.) Once I forgot to return to hardware math mode by copying math.881 into math. I happily used my computer for months before I ran a floating-point program that was so slow I was

convinced that something was broken. Eventually I tracked the performance problem down to the math module and fixed it with a quick copy command. For months I'd been using the wrong math module and no program had noticed. OS-9 users take this sort of interchangeable part for granted. If one of the OS-9 hardware vendors gives us a Weitek floating point chip, we'll expect him to include a math.weitek module.

The OS-9 I/O subsystem also configures itself dynamically. Device names refer to modules called device descriptors. These modules contain various facts about the device, including the name of a device driver module that will handle low-level I/O details, and a file manager that will handle high-level processing. When a program opens a path, the kernel (or IOMan on OS-9/6809) locates the three modules involved and binds them into a team. The I/O situation is fluid. I load the SBF file manager and the driver and descriptor for my tape driver about once a week for my weekly backup. There's no point in wasting memory on the modules the rest of the time. When I'm ready for my lunch break on Saturday, I type

```
load sbm20 sbf mt0
```

then I start the backup. When the backup is done I type

```
deiniz /mt0; unlink mt0 sbf sbm20
```

and the modules are removed from RAM. I use my PC file manager software the same way, loading it when I need it and unlinking it when I'm done.

Loadable device drivers and shared libraries are relatively new ideas in the UNIX(TM) world, and they are painful grafts onto that operating system. For OS-9, they are consequences of the basic design philosophy.

Software on silicon never really made it. Maybe RAM prices fell too fast for it. However, the design features of OS-9 that were motivated, at least in part, by the ROM idea, are still close to the leading edge.

+++

The Motorola 68000 Family

A Historical Perspective

by
Tom Johnson
68000 Applications Engineer
Hi-End Product Marketing
Motorola, Inc.
6501 William Cannon Dr. West
Austin, TX 78735-8598

In the beginning...

By the waning years of the 1970's, things were about normal for the electronics industry, and for the world in general. The war in Viet Nam was slowly fading from memory, Jimmy was president, "Billy Beer" was the rage, inflation was gearing up to double digits, and the largest selling computer was the Digital Equipment Corp. PDP-8. The integrated circuit industry was firmly established, most programs were written in assembly language, and 64k bytes of address space still seemed like a lot. The term "microprocessor" was still new and exciting, and the most sophisticated MPU available was the 6000-gate hard-wired Motorola 6800. Issues of *Radio Electronics* still ran articles on how to build PONG games and "do-it-yourself" 8080-based computers. Microsoft wasn't yet a world power, CPM was the operating system to end all operating systems, and IBM only made mainframes. Process-wise, 5 micron NMOS was the process of choice for microprocessors.

During the fall of 1977, barely 4 years after the introduction of the 8080, in a conference room in Austin, Texas, several engineers gathered together to discuss a "next generation" microprocessor. In attendance were some of the leading microprocessor visionaries: process engineers, design engineers, packaging engineers, software engineers, marketing experts, and one man whose vision would become a legend in the microprocessor arena, Tom Gunter. These people were gathered to talk about Motorola's next step in evolution for the 6800 family. Gunter proposed a 16-bit microprocessor with extensibility to 32-bits — a radical proposal for the time, considering that there existed no market for such a part. Several attendees argued that a processor this complicated couldn't be made from existing technology. Some argued for a code compatible MPU, like the rumored 8088 from Intel. What Mr. Gunter proposed was a microprocessor containing about 68,000 transistors — greater than an order of magnitude more than ever before manufactured. The part would be made in 3 micron HMOS — a process that didn't yet exist! The die would be HUGE — larger than any ever attempted for a commercially produced integrated circuit. The package would contain 64-pins. The clock speed — 4 MHz, single phase! Laughable. The device would contain 32-bit data paths and two separate ALU's. Impossible. And finally, there was the *small* matter of microcode.

You Want to Use WHAT?

Microcode was still a new and unusual beast for microprocessor engineers. True, mainframes had used this technique for a while, and the AMD 2900 required microcode, but no one had ever tried to put microcode on the same device as the CPU. Many said it couldn't be done. The sheer size of the proposed device meant that hardwiring all instructions would be impossible. Microcode would be required. But even using either a traditional single-level, or two-level microcode approach would result in a device too large to be producible. What was required was a totally new approach to the problem. A split-level microstore was proposed, where a first, narrow half, with a fully populated micro-address decoder, would control the subsequent sequence of instruction microaddresses. A second, wider control half with a "sparsely populated" microaddress decoder would actually control the device. The use of a "sparse" decoder allowed more than one microaddress to share the same control state. This sharing of control states between (possibly) many microaddresses resulted in significant silicon savings. Microcoders could now re-use control states for as many instructions as necessary, greatly reducing the overall storage requirements for the microcode, while at the same time ensuring that all possible instruction op-codes were fully decoded to trap illegal bit patterns.

The reception to this idea of Motorola pouring R&D money into a device which couldn't be built would have been cold indeed if not for a few of the other attendees. It had been theoretically proven that the split-level microcode structure could work. It had been proven, again theoretically, that the design could be produced using existing (and some not-yet-existing) computer software. The process engineers were pretty sure that a 3 micron HMOS process could in fact be ready in time to process the design. Packaging said that a 64-pin DIL package, while expensive, was not unrealistic. Fabrication engineers said that, while yield would be low initially, they thought that such a complicated device might be producible. Motorola *bet the farm* that its team of microprocessor gurus could pull off what would be perhaps the most complicated engineering feat of the century — the 68000.

A Blueprint for Success...

What made the 68000 so innovative? For the first time in the short history of microprocessors, software engineers were in on the design from the very beginning. The concept was to create an extensible architectural platform which would allow Motorola to provide new family members with increasing performance and functionality, while at the same time, providing a basis for the easy implementation of high-level language compilers. It was thought at the time, and rightly so, that most future software development would be in high-level languages. If this concept was accepted as fact, then the architects of the new processor should look to those platforms which were already being used primarily for high-level languages — minicomputers and mainframes. In order to provide this high-level language support, while maintaining an open eye to the future, several precepts should be adhered to. The number of instructions should be kept reasonable — this reduces the effort required on the part of the compiler to choose the correct instruction. Special-purpose instructions, and dedicated registers should be kept to a minimum — this allows compilers to allocate registers without worrying about later register conflicts with other instructions. The available addressing mode set should be as rich as possible to give the compiler the maximum flexibility in accessing memory. Finally, the machine should display reasonable orthogonality for addressing modes with respect to register usage, instructions with respect to register usage, and addressing modes with respect to instructions. All of these precepts must be kept foremost, while remembering the limited silicon real-estate available.

Early in the design cycle, it was decided that wherever a trade-off must be made, it would be made in the direction of greater performance rather than greater functionality. This discarded functionality could be added later, if it was still deemed desirable, in future family members. One of the biggest decisions made in the design was the decision to abandon upward instruction-level support for the 6800 family. This was a radical decision. Intel had paved the way in microprocessors, by fully supporting the 8008 programming model in the 8080, and later in the 8088 and 8086. Zilog had produced the Z80 offshoot which still maintained software compatibility with the 8080. For Motorola to not support the 6800 family in the 68000 was almost heretical. All 6800 software would need to be thrown away, and re-made from scratch. Once this decision was cast in concrete, a single concession to the past was made — the 68000 would support the use of 6800 family peripheral devices. Why?

On Re-inventing the Wheel...

In deciding on peripheral support for the new processor, there were basically two trains of thought: one was to hire more design engineers and try to design a complete family of peripherals in tandem with the processor design, the second was to re-use the existing 6800 peripherals. Either path was strewn with pitfalls. If new peripherals would be designed, then literally millions of dollars in existing peripheral designs would be scrapped, and the re-designs would cost the company both time and money. If the second path was chosen, then concessions would have to be made in the hardware and instruction set, using valuable silicon to support older devices. From the vantage point of 1989, the decision seems simple, but then we have the benefit of hindsight. In 1977, the decision was much more difficult. The wrong decision might result in the failure of the product line, the loss of large chunks of revenue for the company, and the firing of many persons. Luckily, the correct decision was made.

To Separate Or Not to Separate...

Many people have commented on the separation of data and address registers in the 68000 user programming model (figure 1). Why was this dual-register approach chosen over a simpler single bank of 16 registers? Three major reasons predicated this design decision. First, a single, large register bank would be more difficult to control due to speed path problems in the register selection hardware. The more registers, the higher the capacitance in the selection logic, and the slower the device would run. Second, the software engineers on the project were convinced that address quantities should be treated differently than data quantities. Addresses are, after all, basically vectors into the memory space, and as such contain both a magnitude and a direction (sign). Operations on these quantities should not set condition codes, and operations on less than the total should maintain both sign and magnitude. This obviated the need for a separate ALU to handle addresses, and thus, separate address registers to simplify the circuit paths. The third reason for separation between addresses and data deals with the number of bits required in the instruction to specify the source or destination register for an operation. With 16 registers, 4 bits would be required for both source and destination — a total of 8 bits, and at least 6 more bits would be required to specify the source and destination addressing modes — a total of 14 bits, leaving only 2 bits left over to specify the operation to be performed. While the 68000 was to have a limited instruction set, it was decided that 4 instruction classes would probably not be enough. Limiting the register structures to 8-each of addresses and data operands reduced the encoding bit requirements to 12 bits, leaving 16 instruction classes. This would be sufficient.

FIGURE 1

M68000 Family User Programming Model

It was determined that future software requirements would slowly gravitate toward 32-bits. The values of ±2 billion allowed sufficient integer flexibility to handle most scientific problems. Thirty-two bits of address allowed plenty of headroom for most foreseeable addressing needs within the next couple of decades. Sixteen-bit integer and address quantities, on the other hand, were already limited by the end of the 70's. Rather than make the processor act on strictly 16-bit quantities, it was decided that for future extensibility and performance, all registers (both address and data) would handle 32-bit quantities, but due to silicon area restrictions, the ALU's would be limited to 16-bits. 32-bit operations would be handled automatically by multiple passes through the ALU's. Future family members could implement all 32-bits of the ALU's, effectively doubling the speed of 32-bit operations. Instructions, and the external data bus would be 16-bits to allow for maximum performance within the packaging limitations, and the external address bus would be limited to 24-bits, although all 32-bits of the program counter would be implemented.

Reaching Outward...

Also early on in the design, it was determined that the external interface provided the designer should be as flexible as possible, reduce the external device count, and take advantage of the latest in high-speed memory designs. To this end, the 68000 was to implement a 4-clock cycle asynchronous non-multiplexed bus cycle. This unique interface would have the processor contain the synchronizing logic, and would allow each type of memory device (RAM, ROM, EPROM, peripherals, etc) to respond in its own optimum period of time — thus increasing overall performance. Additionally, a bus thus structured would easily allow the introduction of "waitstates", something brand new in microprocessors. An interesting side note here is that the 68000's ALU cycles in only 3 clock cycles, while the bus cycles in 4 clock cycles. The 68000's bus could easily have been designed for 3-clock cycle operation (memory access ≈ 105 ns @ 8 MHz), but since few existing memory devices could keep up at this speed, it was determined for marketing reasons to make the bus cycle 4-clocks (memory access ≈ 230ns @ 8 MHz).

...And Upward...

The 68000 introduced several new concepts to microprocessors: separation of user and supervisor code and data spaces, restrictions on self-modifying code, on-chip 7-level interrupt controller with automatic masking of interrupts, on-chip exception processing with 256 exception vectors, fully-trapped op-code map (no more wierd undocumented instructions), etc. The first of these recognized the differences between operating systems and user applications, allowing operating systems to be written which operated in memory areas inaccessible to applications. System-level tasks were given extra resources and instructions also not accessible to applications (figure 2). As new features were later added to the 68000 family, it was this "privileged" set of resources which was modified, allowing all user-level object code to be moved intact to newer family members. The foresight of the architects of the 68000 family can be seen in the lack of crude hacks and special operating modes to support antiquated system-level architectures.

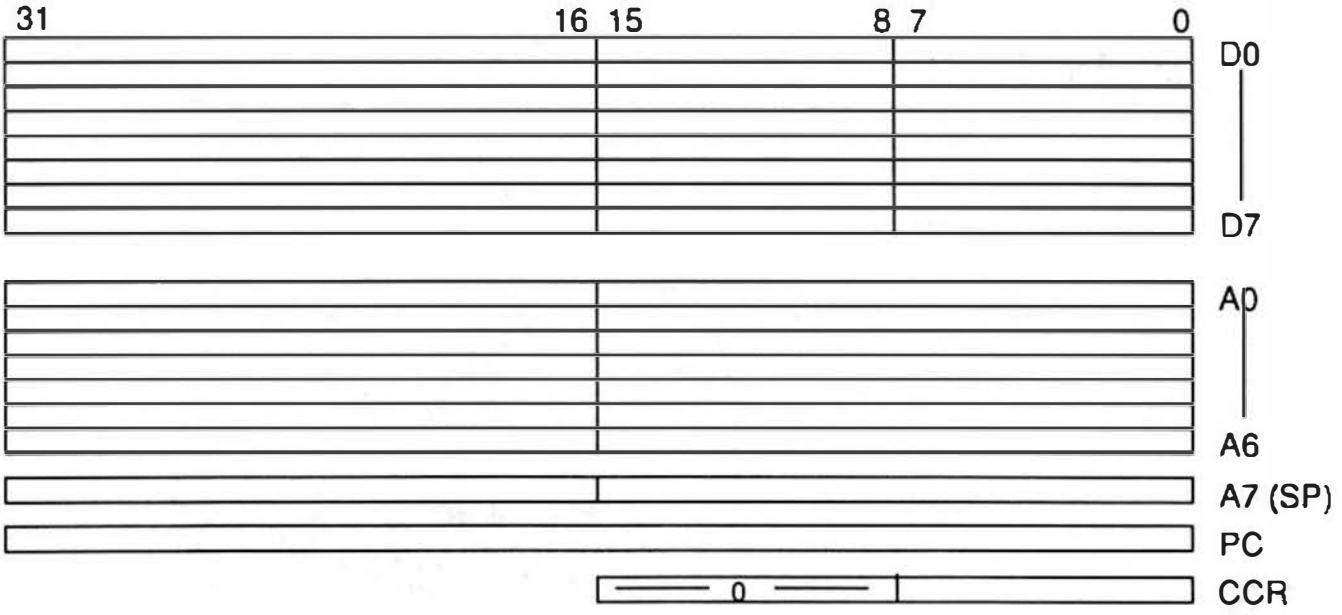


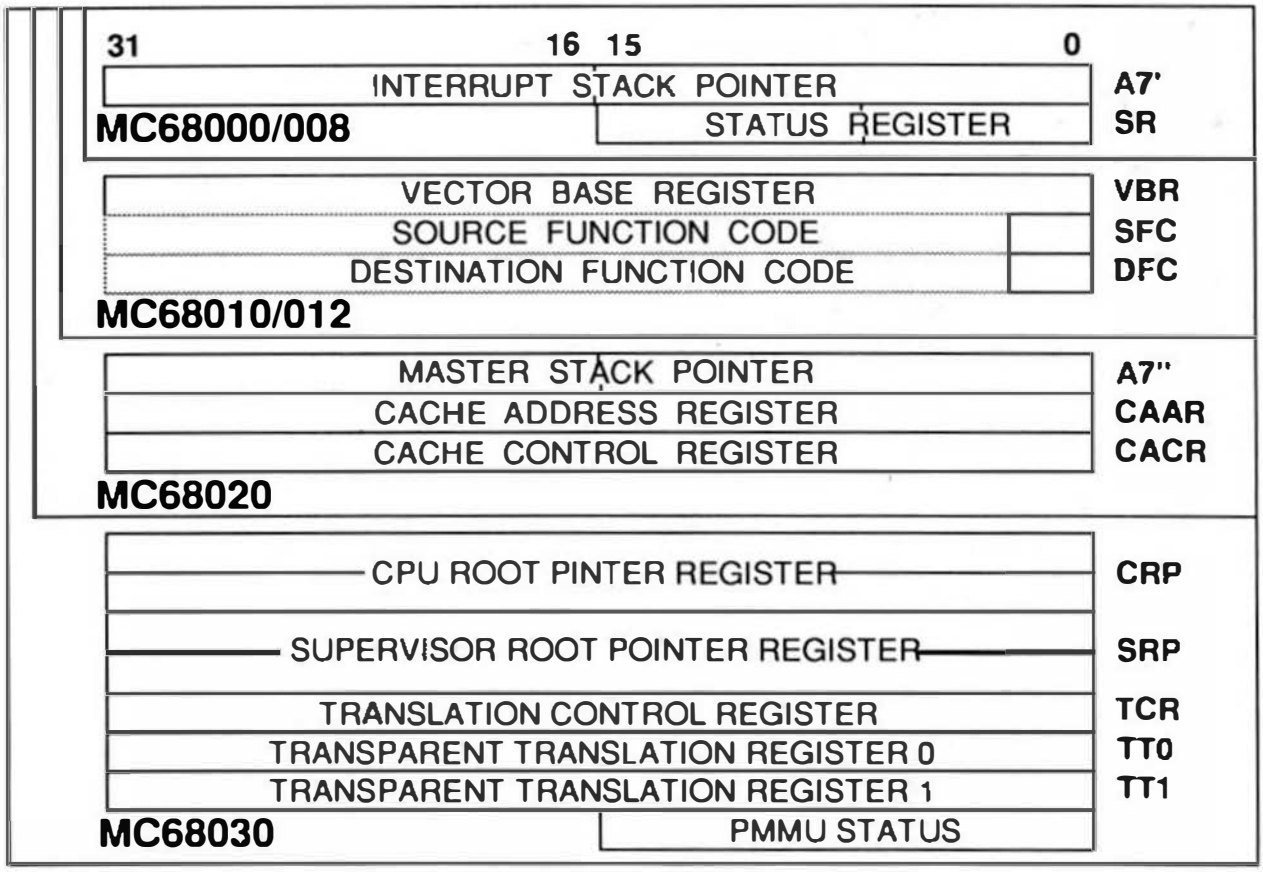
Figure 2

M68000 Family Supervisor Resources

Time Marches On...

The basic 68000 architecture has been enhanced over the years, adding virtual memory and virtual machine support, cache memories, dynamic bus sizing, faster clock speeds (up to 50 MHz), and a plethora of other high-level firsts. Performance has been increased from the "paltry" .35 MIPS of the original 4 MHz 68000 to 12 MIPS for a 33 MHz 68030. Future family members will move into even higher performance areas. Ten years ago, the 68000 set the standard by which all future microprocessors would be judged. Ten years ago, the 68000 introduced the word "superlative" into the vernacular of microprocessors. Today, the 68000 is a simple commodity part, overshadowed by its own offspring, and by ultra-high performance microprocessors like the 88000 RISC and like genre. The 68000 spawned industries never before possible — scientific workstations, windowing systems, distributed processing CAD/CAM, desktop publishing; and pushed others to new heights — robotics, factory automation, UNIX, etc.

It is true that from small acorns large oak trees grow, and only a few attendees at that first meeting in the fall of 1977 had even a glimmer of the possibilities of the new architecture they were about to unleash on the world. The microprocessor which couldn't be built. The Motorola 68000. The 68000 was introduced to the world in July of 1979, and since, over 25 million 68000 family processors have made their way into the mainstream of day-to-day computing. As they say..."the rest is history."



SmartWare Office Automation Package

Microware Systems Corp.
1900 N.W. 114th Street
Des Moines, Iowa 50322
Phone: 515/224-1929

Western Regional Office
4401 Great America Parkway
Santa Clara, California 95054
Phone: 408/980-0201

Microware Japan Ltd.
41-19 Honcho 4-Chome
Funabashi City
Chiba 273, Japan
Phone: 0474 (22) 1747

Microware is proud to announce the immediate availability of SmartWare, a powerful, integrated spreadsheet, data base, word processor and time manager package for OS-9/68000. Licensed from Informix, a leader in SQL and office automation technology, SmartWare has been adapted to the OS-9 environment and works with virtually any ASCII terminal.

FEATURES

- Multi-Level Password Protection
- Linked-Window Scrolling
- Keyboard Macros
- Access to OS-9 without Leaving SmartWare
- Personal "Time Manager" Appointment Calendar
- On-Line "HELP"

Product Overview

The SmartWare integrated software package provides the productivity tools engineers need to write proposals, model new systems, manage information and communicate with others.
Smart is comprised of five integrated software modules:

- Data Base Manager
- Spreadsheet
- Word Processor
- Communications Package
- Time Manager

Each of the individual SmartWare modules is capable of outperforming any stand-alone program in its category. However, SmartWare's real advantage lies in its ability to easily move data from one integrated module to another. All SmartWare packages feature common commands and screen designs to maximize ease of use. A menu umbrella provides a cohesive user interface which ties all modules together into a seamless, integrated package.

Data Base Manager

SmartWare includes a fully relational data base manager. Each data base can contain up to one million records with each record supporting a maximum of 255 fields and 4,096 characters. Several different field types are available including alphanumeric , numeric, date, time, phone, sequential and inverted name. With the inverted name function, the field is sorted on the last word in the field. Up to 15 key fields (with 1,000 characters each) can be defined for each record and a key field can also be used to perform fast searches.

Creating a data base is easy and involves three simple steps:
First, create a file structure by defining fields. Second, create a data entry screen (a default comes with the package). And finally, enter the necessary data. A variable length record option allows SmartWare to only store the data actually contained in each field. Conversely, fixed length fields provide faster access to data retrieved from disk. A password protection mechanism offers secure data control.

Once data is entered, SmartWare provides easy access using queries and custom reports. The query command isolates a subset of the data base file. The query command can be used to select upper and lower field values, relational operators and to set the order of evaluation. Custom reports are readily available using the "report" command. This command specifies fields to be printed and report format. Reports can be structured as either reports or forms. A variety of column definitions and number formatting facilities make reports easy to generate and read.

Spreadsheet

The SmartWare spreadsheet contains a number of features found in other popular spreadsheet software packages, plus many unique enhancements. The capacity of the spreadsheet is 9,999 rows x 999 columns. A "sparse matrix" algorithm is used to save spreadsheet information, minimizing RAM/disk storage requirements and speeding execution. Up to 50 windows are available on any one screen to display consolidated worksheet data. Other spreadsheets may be accessed using formulas in the spreadsheet module.

SmartWare is one of the very few spreadsheets that provide linear algebraic functions. Matrix commands provide the abilities to generate eigenvalues and eigenvectors and to invert matrices using the Gauss-Jordan method. Eight other types of calculation may be used including math, trigonometric, statistical, financial, date, time and regression analysis.

A powerful graphics command can be used to print data in a variety of forms. Two and three-dimensional bar charts, line and scatter plots, plots (bar, line and scatter combinations) and pies (including 3-D) can be easily printed using a dot-matrix printer.

Continued on page 35

Word Processor

The SmartWare word processor was designed to make writing easy. It permits concurrent display or "cut-and-paste" operations from one document to another using up to 50 different windows. The word processor can be used to write a basic business letter, yet provides sophisticated features necessary to create complex technical documentation. Text manipulation features include footnoting, headers and footers, alphabetized lists, auto-hyphenation and spelling checker.

The word processor enhances personal productivity by simplifying text creation/ editing and customizing documents. And as text is inserted or deleted, paragraphs are reformatted automatically. Distribution list mailings and individualized forms can be produced using the merge command.

Communication Package

Hardcopy output can be printed in 12 different fonts (depending on printer support). Other printer functions such as underline, boldface, etc. may also be supported. Graphs created using the spreadsheet module can be embedded in printed documents. Printer control codes allow configuration of unsupported printers.

Transmission and reception of files from another computer is facilitated by using the Communication Module. When using a modem, files can be transmitted between remote sites as text or using the Xmodem protocol. Data can be easily coordinated and shared by multiple systems at various locations. Password protection is available to secure data against unauthorized access.

Time Manager

Meetings and appointments can be scheduled using the Time Manager module. This automatic calendar/diary can maintain itineraries for individuals or functions. Thirteen commands are available and a status line displays the time, date and current file name. A description of up to 25 characters can be displayed in the "meetings" window of the day screen. The "week" or "month" screen displays a schedule with markers indicating meetings or appointments.

SMARTWARE FUNCTIONS

Business Functions	Date Functions
Input Functions	Logic Functions
Numeric Functions	Random Functions
Statistical Functions	Statistical Data Base Functions
Test Functions	Time Functions
Transcendental Functions	Trigonometric Functions
Variable Functions	Miscellaneous Functions

+++

SOFTWARE USER NOTES

A Tutorial Series

By : Ronald W Anderson
3540 Sturbridge Court
Ann Arbor, MI 48105

From Basic Assembler to HLL's

6809 and 68000 Compared

After a lot of work, there are now two current versions of the editor PAT. First is the new SK*DOS version written in Whimsical, and second the 6809 version freshly updated to do everything the other version does. It is written in my old favorite PL9. The two versions work just about identically. The final object code byte count and some statistics about them may be interesting to most of you. The 6809 version is 18,148 bytes and the 68K version is 19,800 bytes. The difference is just about 9%. The 68K version can edit a much larger file. The other night when the 512K memory expansion for the PT68k-2 arrived, I made the edit buffer 500,000 bytes and found that it still worked fine. Of course addressing any range greater than 65K requires a larger address. That is, the address must be greater than 16 bits. Therefore one would expect that more code would be required to do the same functions. Perhaps the interesting thing is the code generated per line of source code. The 6809 compiler PL9 generated 6.39 bytes per line and the 68000 compiler, Whimsical 6.55 bytes. The line counts include blank lines between procedures and comment lines. The bytes per page count is 363 for the 6809 version and 375 for the 68K version.

More on Assembler

I've been spending some time trying to write some utilities a little beyond the single pagers that we all start out with. Last month's ADDRESS utility was just a little larger, so it qualified. Let me urge you SK*DOS users to sit down and TRY writing a few little programs. There have been examples here in this column and others. What brought this on is the arrival of Motorola's 68000 Audio Cassette course which they are presently offering for $125. It is about 4 1/2 hours of cassette lecture with extensive notes in sort of a workbook. I was primarily interested in learning more about the hardware end of the 68000 and along with that, some of the more advanced topics such as exception processing and interrupt handling. The course arrived a couple of days ago and I've gotten to the half way point. It is nicely done but it strikes me that it sounds incredibly detailed and complicated just listening to it, but it is not nearly that hard to understand when you launch out and start using it.

I had started reading books on "Programming the 68000" three or four years before I had a 68000 based system in front of me. I never got very far with the book because I got lost in detail and had to pinch myself to stay awake trying to read through the addressing modes and the instruction set. I was very familiar with the 6809 but this looked so different that I would never be able to grasp it (I thought). Honestly, with the SK*DOS manual (read it first) and the Motorola 68000 User's Manual (ditto), it took a very short time to see that the instruction set and the notation used is quite straightforward.

I don't claim to be an expert assembler programmer with a 68000 yet, but I can write programs and make them work with a lot less effort than it took to get the first 6800 and 6809 programs running. Of course some of that is attributable to experience at assembler programming and/or debugging programs in general, but certainly a good part of it is the simplicity of the structure of the instruction set. The instructions are uniform and behave as you would expect them to

behave. If you try to use an instruction that is not implemented (i.e. an addressing mode that is not legal for the particular instruction) the assembler flags the error and you refer to the User's Guide and figure out another way to do the desired function.

There are a few things that are not nice about the 68000. The designers made a decision for us as programmers that any data that is imbedded in the program code must be a constant and can only be read via pc relative addressing. That is, if you have a program followed by a couple of DC.W declarations to set aside space for a variable, you can do the following:

```
move.w var(pc),D0
```

That is, you can read data from the location var(pc). Pc counter relative addressing mode is used and the value is moved to D0 in the above example. Now you can change the value in D0 but you can't write it back as below

```
move.w d0,var(pc)
```

This is not a legal instruction in the 68000 unfortunately. It takes two instructions to put the value back into the variable.

```
lea var(pc),An
move.w D0,(An)
```

That is, you have to load the pc relative address of the variable into an address register (any of them that is not being used for another purpose) and then move the data via an indirect addressing access. It is too bad, but the designers decided that all data (i.e. variables) for a program should reside in another block of memory somewhere. Indeed the hardware of the 68000 is set up so that the user can decode the three function code lines and prohibit writing to the program section of memory, allowing writing only to the Data section. Peripheral Technology has chosen not to do this, and programs written to run under SK*DOS, though they must be position independent, can put the variables just after the program code (or before it, or within it, between subroutines or after an unconditional branch to elsewhere). The only complication is that you have to use two instructions to write to a variable as outlined above.

Modula-2 for SK*DOS

A week ago, a package arrived in my mail. It turned out to be a complete Modula-2 compiler implemented for SK*DOS 68K by Mike Randall of Wellington New Zealand. Mike said that his effort was about complete and ready for some serious testing so he had sent me a copy to try. Since I have been a fan of Pascal and its derivatives I would have been happy with a Modula-2 compiler to run on ANY computer. Having one for the 68K system was almost too much to bear. The documentation was primarily the specifics of the implementation and they stated that they are not a tutorial on Modula-2.

Mike recommended two books on the subject, the first by Professor Niklaus Wirth. We had obtained Wirth's own book on Pascal several years ago from a publisher in Switzerland. It took a couple of months to get one. I went to the local shopping mall which has both a B. Daltons and a Walden Books store. Though I found several feet of shelves in both stores devoted to running AutoCad, or how to use Lotus 1 2 3, and surprisingly, about 6 feet of books on "C", I found NO books on Modula-2 in either store. The package had arrived on Saturday and I had managed to get to the bookstores on Sunday.

I figured my only hope would be one of the bookstores on the campus of the University of Michigan. (Living in a University town has its advantages at times). It was Tuesday before I found the time to call the bookstore and inquire about Modula-2 books. I was told that there were FOUR different books in stock. I chose "Modula-2 Made Easy" by Herbert Schildt, published by Osborne / McGraw Hill.

That night I set out to read the book, and by the time I had gotten a few chapters into it, I ran the compiler to compile a first program. It worked right off, and I was impressed with how close to the "standard" the compiler is. Over the next few days I found a couple of minor bugs which have been reported to Mike by International Express Mail. I managed to get my scientific functions package translated into a Modula-2 Module.

Let me give you some first impressions of Modula-2. First of all, I was impressed, no, actually I was more amused at how much Professor Wirth was influenced by "C". Pascal was a language written to teach students how to program properly. As such it was done in the days of batch processing and it lacked any standard way to access disk files. The details of that were left to the implementors of Pascal, and therefore each Pascal compiler had a little different flavor when it came to file handling. Apparently Professor Wirth liked the idea of a "standard library" as used in "C", so he left all the system dependent things like file accesses out of the language proper, and put them into a "standard library". Some of the modules in the library are "Terminal", which contains character and string input and output routines; InOut which contains those plus file opening, closing, read, write; RealConversions, containing RealToString, and StringToReal conversions; RealInOut, containing ReadReal, WriteReal, and for some reason WriteRealOct, a procedure for writing a real number out in Octal. I replaced that with a WriteRealHex, essentially rewriting that module for myself. There are several others, but you get the idea. When your program or module needs to use one of these, you "IMPORT" one with a statement like:

```
FROM Terminal IMPORT
Write,WriteLn,WriteReal,ReadReal;
```

When you import a procedure from any module, that whole module gets linked in as part of your program. Some of the modules link in other modules necessary for their operation. I wrote a three line program approximately as follows:

```
WriteString("Hi there");
WriteLn;
number := 2*PI;
WriteReal(number,12,7);
```

Remember that I had written my own WriteReal, and it is a little different than the standard library version, but made to be more like the Pascal version with regard to the field specification for the number format. At any rate, I imported WriteString and WriteLn from InOut library, and that "dragged in" all the file handling procedures which I didn't need. WriteReal was imported from RealInout which loads RealConversions. All in all, I had 68 sectors of code, about 17K for that four line program. I wasn't very impressed. Then I found that WriteString and WriteLn were also in the Terminal library, and that didn't pull in all the file handling procedures. I made the change and recompiled and linked the program and found that I now had 36 sectors of code.

I tried an equivalent program in "C" using printf() to print the string and the number, and the program was 26 sectors long. I tried the same in Whimsical and I got 6 sectors of output code.

I was puzzled at the seeming inefficiency of code generation when it dawned on me that both "C" and Modula-2 were pulling in whole libraries for the need of one function. Whimsical has the in/out functions built into the compiler as separate subroutines that are included if needed. You don't get any more code than you need.

I can do something about the problem in Modula-2 simply by getting my hands on the source code for the libraries. It sounds as though they might be described in Professor Wirth's book. The libraries could be split up in such a way as to allow importing only code that is needed. I expect the efficiency of code generation would go up considerably at the expense of having a library of standard procedure modules, a large number of them, requiring a lot of bookkeepping to get them all straight.

After some thought, I have a question for all the compiler designers and writers out there reading this (all two or three of you, that is). "C" in my mind generates an outrageous amount of object code for a simple program. Now I think I understand why. "C" generates Assembler source code. Why couldn't an optimizer be written that would operate on the assembler source code to eliminate all code that is never run in a program. Any complete unit of code, (something that starts after an unconditional branch or RTS instruction) with an entry label that is not referenced anywhere else in the program, down to an RTS or label that IS referenced somewhere else in the program, or the program end, could be deleted from the assembler source code simply by deleting such sections. The fact that the program uses labels at that point rather than

offsets or absolute addresses, would mean that such code could simply be deleted and the remaining code, all of which is executed by the program, would still assemble correctly.

Oh, I realize at least in the case of the "C" compiler that the whole assembler source file is not pulled together as the output of the compiler. There are lots of "lib" references in the output file, but those portions of code COULD all be pulled into one big file that could then be handled as above. I tried my hand at it manually and soon discovered that it was too hard to keep track of labels manually, and that it was really easy to delete one line too many so the program wouldn't compile.

In the case of my tests above, about 85% of the first Modula program and about 70% of the "C" program was code that was never run. Wouldn't it make more sense to work on an optimizer that would remove code that is never used by the program than to make an optimizer that simply removes consecutive instructions like:

```
LEA A0,VAR(PC)
MOVE.W D0,(A0)
MOVE.W VAR(PC),D0
```

This trio simply stores the contents of D0 and reads it back from memory, and the removal of such instructions is typical of what most optimizers do. The optimizer might assume that the contents of D0 need to be stored in the variable, but they are already in D0 by the third instruction, so that can be deleted. By actual test, the optimizer pass of the OS9 Microware "C" compiler reduces the code by some 3% or so. One that removed non-accessed code could reduce the program size by 75% or more in some cases.

I should point out here that my test program is really a worst case. I purposely used string and REAL functions so there would be a couple of diverse I/O routines included in the program code. In practice with non-trivial programs, you would almost surely use more of the library functions that were included, and the code that you write, of course would expand while the library code would not. The differences would be smaller percentages than with the very simple program.

Actually, the optimizer idea could be made to work on binary object files too by disassembling them, optimizing and then re assembling them, but there would have to be a lot of careful work done identifying data tables and making sure the disassembler didn't generate invalid labels.

The optimizer would have to make multiple passes because eliminating unreferenced code might eliminate references to other code which could be eliminated on another pass etc. The program would have to run until the program got no smaller on a pass. It could be a fairly long process, but then as with all optimization programs, it wouldn't have to be run until after a program or module is debugged. Am I dreaming or is all this possible? Or would it be much simpler to have a smarter compiler or linker that could link in only procedures that are referenced? Or have I lost all of you?

Well, anyway, Modula-2 looks like a nice language. Like Pascal it is restrictive, particularly with regard to variable typing. However, unlike Pascal it has a SYSTEM library with some variable type definitions that make life easier when you want to "bit fiddle". That is, if you want to get in at a very low level and manipulate bits as in a math operation. In the scientific function calculations, it turns out that a great improvement in their execution time and accuracy of the results can be obtained if one can get into a REAL number and extract and manipulate the exponent. (A simple example would be to divide the exponent of a number by 2 to get a fair first guess as to its square root, and then use Newton's method to refine it. As you might imagine, manipulating the exponent is crucial to EXP and LOG functions as well. I was able, using the SYSTEM functions and variable definitions, to write procedures to extract the exponent of a real number, and to put an exponent into a real number to replace the existing one. Once those were done, the scientific function package was easy to implement.

The caution about using things from the SYSTEM library is that the code now might not be portable to another system running a different modula-2 compiler. Actually, the code that I wrote would run in another system with a different compiler only if the REAL number representation were the same. However, at

times it is worth making the program non-portable for an improvement in performance. The two procedures in question could be commented well to show what they did, and the "porter" would have to rewrite them to work with a different number representation.

More Modula-2

As Gomer Pyle used to say, "well Gahhhhh-ley" It seems that I am about a compiler generation late. I've spent a couple of weeks getting my thinking switched around from Pascal to Modula-2 (I skipped right past Modula, and Algol was before my time in computing), and now I find that Professor Wirth has said that Modula-2 is obsolete. See Bud Pass' column in the April '68' Micro Journal forthe update. I'mnotkidding at all, I did write the above discussion of Modula-2 BEFORE I saw Bud's article. I certainly did smile at his mention of how Wirth is coming around to writing compilers that look more and more like "C". That observation certainly is in line with what I said above.

Of course computers have changed in character a great deal since the days when Pascal was written, so of course the philosophy of programming must change also. Modula-2 fixed the problems with Pascal in the area of defining file handling procedure and a number of other places. To my thinking, it became a practical language to apply to real world applications other than pure computer science research and running database software on mainframes. As I just mentioned above, Modula-2 had a real way to get at the hardware. The absence of that facility in Pascal is probably its greatest drawback with relation to "C". Sure, a "C" program can be very portable, but if a programmer wants to get to the hardware, he can do it very easily with a pointer. Modula-2 added that possibility with the type definitions and operations in its SYSTEM library.

Now Professor Wirth has decided that the language was contaminated by those facilities because they allowed a user to write non-portable code, and therefore has thrown them out in his newest creation Hermion. Professor Wirth, wake up. Look around. Where are computers going? Our house has a microprocessor based microwave oven. Two of our cars are controlled by at least one microprocessor, probably more like two or three. The gas range has one, though a simple one, in its timer and control section. The video cassette recorder has one so that it can be programmed to select a channel at any time within the next two weeks and record a program unattended. My wife has a computer controlled sewing machine that can do many fancy stitches and even machine embroider names or initials in two or three different font styles. A friend of mine, one morning, pulled into a gas station but couldn't buy gasoline because "our computer is down". The comer video rental store uses bar code readers to read the video cassette labels and the customer's "membership card", and a computer prints out the sales ticket. The supermarket reads bar codes on packages with a laser bar code reader and the computer totals the bill, printing an itemized list of the articles purchased and their prices.

That only is a small list of the commercial use of computer chips. The industrial uses have taken off also. A whole class of programmable controllers has replaced relays in machine control applications. Several months ago one afternoon I was beginning to debug some software in a large machine. I was pushing the "jog" button on a large control panel and trying to get a servo motor to tum a fraction of a revolution each time the button was pressed. It occurred to me rather suddenly that there were three computers between the push button and the servo motor. The control panel was being scanned by an Allen Bradley PLC, a programmable controller that is programmed in terms of "ladder logic" the symbolism used by relay control logic designers for many years. That programmer contains a microprocessor. It was scanning inputs and seeing the contact that I was closing by pushing the button. It performed some logic and output a signal to the 6809 computer supplied by my company as part of the machine. The 6809 computer responded to the signal from the PLC by generating an ascii string of characters output on a serial port to tell a servo motor controller to tum the servo motor 1/10 revolution clockwise. If all went well, the servo controller did that and sent an ascii string back to the 6809 to say that the command had been executed. The 6809 software contained a timing loop so that if the servo controller didn't respond in a timely manner, the screen of the computer display indicated Servo Controller Not Responding or some such error message.

Admittedly, the computers involved are small, but there is a lot of software there, and those three computers all had to be programmed by someone. There was a lot of hardware interfaced to each of the computers also. Two of them had no standard input or output devices connected to them. One had a CRT monitor.

The point is that far more computers are finding their way into dedicated applications like these with hardware interfaces involved than into traditional computers these days. Are the programmers of all these applications forever to be forced to program in Assembler for the sake of the purity and portability of a computer language? Ask the programmer of the Pfaff sewing machine's computer if he cares whether or not the code is portable, and he will probably say that he certainly does care, the less portable the code the less likely a competitor is to "port" it over to another processor!

Modula-2 solved the problem to a large extent, though access is still a bit awkward. Purists writing code for computer systems can choose not to use SYSTEM, and thereby maintain the portability of their code, which is a great advantage in broadening their market.

Those of us who are writing programs to run hardware for industrial control or automotive applications are looking for efficient code and easy interface to hardware. Give us a way to declare a variable AT an address and further to declare a real, a long word, two words, and four bytes all at the same address. One can write a "pure" square root routine, but it can be done in such a way as to run much faster by extracting the exponent byte of a real variable, manipulating it and reinserting it. Take all of our shortcuts away from us and we will all flock to something else in the way of a language, for example, my favorite so far for the 68000 system, called Whimsical. Whimsical lets me get at the bytes of a real number. It uses an IEEE real number representation in which the high order bit is the sign of the mantissa, the next 8 bits are the 2's complement exponent with an offset of 127, and the remaining 24 bits (the high order bit of the mantissa is assumed always to be a 1 since numbers are stored as binary fractions normalized so that the highest order bit is always 1), are the mantissa. It therefore is "hidden" under the least significant bit of the exponent.

To get the exponent one has to grab the high order WORD of the REAL, shift it left once, and take the high order BYTE of that. Whimsical has the type SHORTINT which is a signed 8 bit integer, that just fits the exponent value range. The exponent can be extracted simply by:

```
SHORTINT PROCEDURE GETEXP(REAL NUMBER);
BEGIN
  RETURN SHORTINT (BYTE[1](WORD[1] NUMBER)<<1)
-$7f);
END;
```

Whimsical uses typed procedures (as "C" does). The procedure is passed a real number as a parameter. The procedure shifts the high order word (WORD[1]) of the real number one place left. It takes off the top byte (BYTE[1]) which is the exponent, subtracts $7F from it (the types BYTE and WORD are compatible with Hexadecimal values). It is then converted to the signed SHORTINT type and returned to the calling procedure. Inserting an exponent in a REAL is a little more complex, but not much. Those two functions are the basis for a scientific function package.

I don't suppose there is much chance that Professor Wirth will ever see this discussion, but I'd be interested in his thinking on the subject if he does.

+++

FORTH

A Tutorial Series

By: R. D. Lurie
9 Linda Street
Leominister, MA 01453

A (TEMPORARY?) CHANGE IN PATH

Starting with this column, I will be spending most of my time with OS-9 and FORTH09, instead of FLEX and FF9. This does not represent a conviction that there is something wrong with either FLEX or FF9, but, rather, a bowing to the inevitability of OS-9! However, I don't plan to abandon all of you who have been with me over the last two years, since I expect that most of what I have to say will still be relevant to any sort of FORTH, at least, I hope so. Furthermore, I hope that this shift toward a greater emphasis on OS-9 will be of some use to the 68000 community, even though I do not have a 68000 machine. This change in emphasis was made possible by the introduction of FORTH09, a product from D. P. Johnson. I don't want to repeat my review of FORTH09 which appeared in the January, 1989, issue of 68' Micro Journal, but I do want to say that Dan has done an excellent job of fixing the few bugs that anyone has been able to find in it, and I can give it an unequivocal recommendation. Since it works with Level I or Level II OS-9, just about anyone with a 6809 can use it, so I have no qualms on that score. Furthermore, I understand that Dan is working on a 68000 version of FORTH09.

In fact, I am so pleased with FORTH09 that I have quit work on my C3FORTH for the CoCo3, since I no longer need it.

OS-9 PATCH

I never thought that I would be offering a patch for OS-9, even one as simple as this, but here it is.

```
l cc3io
c 05ac 18 1d
```

This patch is called by modpatch from my startup file. It changes the shifted-left-arrow from CAN (^X) to the harmless GS (^=). I needed this change because I was continually fouling up with the FORTH09 editor when I reached for the shifted-right-arrow, which functions as the TAB key. CAN erases the current line, so you can imagine my frustration and language when I accidentally hit the wrong key!

Output Directly to a Text File

I thought that I would show off a few features of FORTH09 by presenting screens 0-7, which were printed with QL directly into a text file by a variation on the technique used in XFRS+. QL is a part of a screen file named "forth.list" and XFRS+ is part of a screen file named "forth.xfr". The command lines were:

```
USING /D0/FORTH.LIST
0" ql.list" >PATH 0 3 QL >SCREEN
USING /D0/FORTH.XFR
0" ql.list" >PATH+ 0 3 QL >SCREEN
```

The first command switches to "forth.list" on /d0. The second line opens a new file named "ql.list" on the current working directory, which happens to be /d1, but it could be anything. The "forth.list" file is then written to "ql.list" by QL, and >SCREEN switches output back

to the terminal, which happens to be /w1. The third command switches to "forth.xfr" on /d0. This time, the last line opens the old file named "ql.list" and uses QL to add its output onto the text already in the file. (Later, I used the OS-9 editor to rationalize the screen numbers so that they would be easier to talk about.)

Explanation of "forth.list":

The first thing to notice is the listing of Screen #0. QL did the expected and chopped off the unnecessary lines 8-15. However, I did want to keep the "blank" line #1, just to make the screen easier to read. Therefore, I put a \ way out toward the end of the line where it would be out of the way, but still would force the line to be printed. I used this trick in all of the screens where it was appropriate. Remember, this is nothing but a space-filler and has no effect on the compilation of the screen.

Line #0 of Screen #1 is particularly important! You will remember in my review of FORTH09 that I mentioned the existence of a PRIMARY and a SECONDARY dictionary. SECONDARY definitions can use PRIMARY words, but not the other way around. Therefore, since the next few screens contain words from the SECONDARY dictionary, they should, themselves, be put in the SECONDARY dictionary. (There are exceptions to this, and I will discuss them later.) In any case, I took the first possible opportunity to select the proper dictionary for the following definitions. This selection holds until it is explicitly changed.

The first definition on the screen uses the OS-9 utility to print the current date and time, using the SHELL command. SHELL causes the OS-9 shell to execute the command string initiated by CR" and ended by " . SHELL can be used to execute any legitimate OS-9 command from within a FORTH09 program, so it cheerfully allows you to crash the system if you are not paying attention! As they say in the monster movies, "You have been warned!"

The definition of CLS is routine, but the use of EQU in Line #8 to define FF (form feed) is unique to FORTH09, as far as I know. EQU creates a synonym using the same execution code for both words, thereby potentially saving a lot of RAM. By the way, EQU is the SECONDARY word I was anticipating in Line #0 of this same screen.

You may wonder why I wrote a new definition of list, since there was a perfectly serviceable one already provided with FORTH09. Well, there were two reasons. In the first place, I tend to accumulate listings of the same screens as I edit them without ever bothering to throw the old ones away. Mainly, this helps me to return to the old form of a definition if I find that I have been working down the wrong path; therefore, I need the date and time stamp on the listing so that I can keep them in the proper order. Furthermore, I really do have trouble following a nearly blank line across the screen or the page to where the number is listed, so that the supplied version of LIST was giving me some trouble in identifying line numbers; my eyes just aint what they used to be! There is nothing notable about this definition, so I won't spend more time on it.

On the other hand, .TRIO resolves a pet peeve with the utility commonly known as TRIAD . .TRIO lets me start printing with any screen number, while TRIAD requires that the first screen number be divisible by 3. Far be it from me to argue with people who are so well organized that their listings always begin with the proper screen number for TRIAD , but I am not nearly so well organized. I think that TRIAD appears only in FIG-FORTH, but others may have also had my trouble. In any case, .TRIO is not a particularly remarkable definition, so I will go on.

The definition of QL in Screen #3 is pretty much as I have shown it before, except for the use of .DATE-T . Notice that this definition uses nested loops, which means that the counter for the outer loop is called J and the counter for the inner loop is called I . The IF ... ELSE ... THEN statement in lines 7-10 determine whether or not there is any text on a line; therefore, whether or not it should be printed.

Explanation of "forth.xfr":

Screen #4 introduces the "forth.xfr" listing, and is normally identified as Screen #0.

Line #2 of Screen #5 has an interesting feature unique to FORTH09. About half way across the screen is the

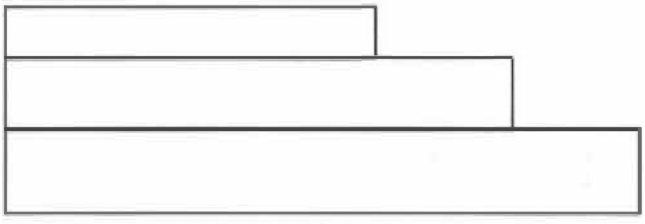
command REMEMBER XF . This puts a marker into the dictionary so that later typing of XF as a command will cause XF and all following words to be dropped from the dictionary. It is a convenient way to clear clutter from the dictionary after it is no longer needed.

XFR+ attaches a single block to the end of an existing file. The entry parameters are set up for the convenience of the operator, in that they follow the FORTH convention of "from" as the first parameter and "to" as the second parameter. SCRN is a VAR which is identical to the VARIABLE SCR ; they even use the same storage bytes. The phrase TO SCRN is the way the screen number is stored in SCRN ; either the word SCRN or the phrase SCR @ would produce the same value on the Data Stack. When I wrote this definition, and the following two, I was too lazy to juggle the values on the Data Stack, so I settled for storage in a VAR instead. This simplified the programming so much that I think that I just had a revelation; namely, don't use the Data Stack in situations where variable storage would be easier! Why did it take me so long to see this?

XFRS+ is simply a looping structure containing XFR+ . Again, the parameter order follows FORTH convention of "from", "to", and "count". The VAR named ADR (line #2) is used to store the pointer to the string so that I did not have to worry about keeping the Data Stack untangled. Line #8 prints the number of the next screen to be transferred, so that I can keep track of the progress of the operation. The name simply means "XFR Several".

XFR does the same thing as XFR+ , except that it also creates the file. As a result, only one block can be transferred to a new file using this scheme; all further blocks must be transferred with XFR+ or XFRS+ .

I use these words to transfer definitions from my "general purpose definitions" file, rather than have to retype them each time I need to reuse a particular block.



FLOATING POINT MATH

Recently, I have been experimenting with the conversion of some engineering programs originally written in BASIC into FORTH. There is really no problem with converting most of the program, but the math operations kill the whole project.

I have found that, most FORTH programming experts to the contrary, many engineering calculations need the flexibility of floating point. Even with 64-bit intermediate products, integer math is just too much of a hassle to be worth the effort! I will grant that any calculations can be done in integer math, if you know what range of numbers to expect as input and output, but integer math just does not have the dynamic range to be used as a general purpose package.

In order to avoid reinventing the wheel, I looked around in the public domain for FORTH floating point packages, but I did not find very much! It is understandable that an author would prefer to sell a complete floating point math package, rather than giving it away, but that does slow down the progress of FORTH as a generally accepted part of the family of languages.

The best that I could find was a file called ZENFP.4TH on either the DDJ or the CLM forum on CompuServe. Unfortunately, my lousy record keeping prevents me from being more informative. This is a package written by Martin Tracy. I had to modify it a little bit to get it to work for me, but the changes were minimal.

There is no question that this package works, but it produces results about equivalent to what I used to get with my wooden slide rule. You can't trust the answers beyond 3 significant figures, and it prints out a lot of trailing zeros, which give the impression of great accuracy, but are really nothing but dangerous junk!

I guess that I will just have to write my own package, if I want something useful. I found a very complete BCD package, written in assembly language for the 6800 in a government publication. I think that it could be adapted without too much difficulty, and I will give it a try. However, it will be a while before I can publish any results, since I have not yet started and there is a lot of testing to be done.

You may well ask why I would bother to do a software math package when I already described an interface to a hardware (obsolete) calculator. I confess to altruism, modified by some self-interest. I no longer use the S-50 machine, so that is not readily available, and it is too much trouble to even try to interface the board to the CoCo3! In effect, I want a math package which will work on any of my machines, including the (shudder) CP/M, so the only solution is high level FORTH. I'll let you know if and when I work it out.

An Apology

I hope that you will excuse the misspelled words and similar errors which cropped up more often, lately. I have been trying to use the Stylo package for FLEX on the CoCo, and it has been a real trial. My main problem has been that I sometimes type faster than the keyboard can respond, so that characters got left out. Also, I do transpose characters, even when I know better. When one adds these problems to just plain poor spelling, errors were bound to slip by even the most careful proofreading. My FLEX assembly language spelling checker does not work with the Stylo file, so I was reduced to visual inspection. As everyone should know by now, 100% visual inspection is an oxymoron!

However, this problem should now disappear, since I have just got a new word processor specifically for the CoCo3. It is called "Simply Better", and believe me, it is! These files can easily be transferred to FLEX for my old spelling checker, or I may start using Spell 'N Fix.

Some Industry News

I thought you would like to hear about a new application for FORTH and the 68701 (a version of the 6801). At the local FIG chapter meeting in January, Fred Carter described and demonstrated an instrument for measuring the light intensity at the end of a fiber optics cable. It is a hand-held device, about the size of a large 4-banger calculator.

The device is mostly CMOS, except for the LCD display. The whole unit is driven by nicad batteries, except for the "ROM", which is actually a CMOS SRAM with its own lithium battery. The RAM draws so little current that the battery is expected to last for the full 5-7 year anticipated life of the instrument. This is the mass-storage system!

The FORTH is multitasking and interruptdriven. There are five to seven tasks to be performed during a measurement, and they have different priorities, so using interrupts was the obvious way to go.

This was a very impressive demonstration!

TIPS FOR BEGINNERS

Brodie and other authors make so much to-do about the Data Stack that many beginners think that real FORTH programmers don't use variables. Actually, judicious use of variables can speed up the programming cycle and certainly make later maintenance and, horrors, changing the program a lot easier for yourself, or for someone else seeing it for the first time.

There was a message thread on the GEnie FIG forum which made a very good argument for never using words which accessed the Data Stack any farther than the third item. In other words, ROT is ok, but stay away from PICK and ROLL . The idea is that it is too easy to get lost in the stack or to mess it up for other words if you have to go that deep.

Variables don't take up very much memory nor execution time on a 6809, so the question of when to use variables and when to use the Data Stack should be a question of which is easier to use and which is easier to understand in the particular situation. Any time you have to use a stack diagram in order to understand a definition, you have gone too deeply into the Data Stack. Remember, at some time or another, you will probably want to examine one of your previously written definitions, and you sure don't want to hassle with following umpteen items on the Data Stack while you are trying to follow the logic of your opus.

I looked back through some of my old code, and I found an interesting pattern. When I first started with FORTH, I used a lot of variables, because I didn't really feel comfortable with the Data Stack. Later, I began to put

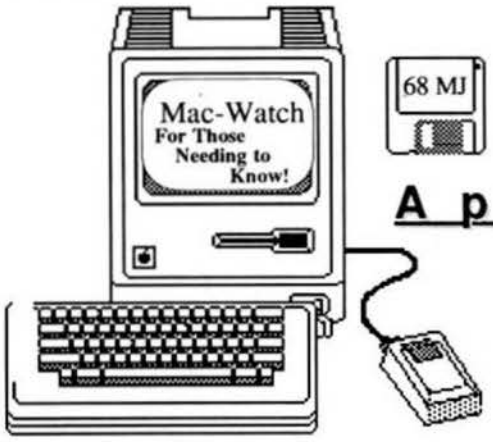
more and more on the stack and less and less into variables as I began to understand how the stack worked. But, now, I just don't want to bother with juggling the stack, so I use more variables. This month's code is a good example of that; a year ago, I would not have even considered using words like ADR in Screen #6. I would have avoided it just because I would have considered it poor FORTH practice!

In defense of the Data Stack, I must point out that it affords secure local variable storage; whereas, any word defined as a VARIABLE (or VAR) must be global (in "standard" FORTH) after it has been defined. As a result, you can sometimes run into the problem common in BASIC, etc. of finding that a variable has unexpectedly changed. FORTH does have ways to force a VARIABLE to act only locally, but that is another topic which I will defer until another time.

Now I know that good FORTH practice is what makes a program run well, while being easy to write and easy to understand!

```
SCREEN #0
  0 FORTH.LIST is a new version of the LIST command.
  1                                                    \
  2 It also contains definitions for:
  3     .DATE-T
  4     CLS
  5     FF
  6     .TRIO
  7     QL

SCREEN #1
  0 SECONDARY  DECIMAL
  1                                                     \
  2 : .DATE-T   ( - )                                  \ RDL122688
  3     CR" DATE T"  SHELL ;
  4                                                     \
  5 : CLS   ( - )                                      \ RDL011289
  6     12  EMIT ;                                      \
  7                                                     \
  8 EQU  FF  CLS                                        \ RDL012589
 10 ->

SCREEN #2
  0 : LIST  ( n - )                                    \ RDL121988
  1     CR    ." Screen #"
  2     ( n )  DUP  .
  3     ( n )  BLOCK
  4     30  SPACES     .DATE-T
  5     16  0  DO
  6       CR    I  3  .R    SPACE
  7       DUP  I  64  *  +  64  -TRAILING  TYPE
  8     LOOP
  9     DROP
 10     CR ;
 11                                                     \
 12 : .TRIO ( n - )                                    \ RDL011289
 13     ( n )  DUP  3  +  SWAP  DO
 14       I  LIST
 15     LOOP ;       ->

SCREEN #3
  0 : QL    ( n1 n2 - )                                \ RDL020189
  1     CR    30  SPACES     .DATE-T
  2     ( n1 )  ( n2 )  1+    SWAP    DO
  3       CR    ." Screen #"    I  .    CR
  4       16  0  DO
  5         J  BLOCK    I  64  *    +   \ point to start of line
  6         64  -TRAILING    DUP        \ chop trailing blanks
  7           IF    I  3  .R            \ if anything printable,
  8             SPACE    TYPE    CR     \ print line# & text
  9           ELSE  2DROP               \ else skip line
```

```
 10          THEN
 11        LOOP
 12      LOOP ;

SCREEN #4
  0 FORTH.XFR transfers one or more blocks from one file to another.
  1                                                              \
  2 XFR+  transfers a single screen to an existing file.
  3 XFRS+ transfers a series of screens to an existing file.
  4 XFR   creates a new file and transfers a single screen to that
  5       file.

SCREEN #5
  0 \ XFR+                                                  RDL012789
  1                                                        \
  2 SECONDARY  DECIMAL      REMEMBER  XF
  3                                                        \
  4 : XFR+  ( scr# adr - )                                 \ RDL121988
  5     SWAP  TO  SCRN
  6     ( adr )  >PATH+                 \ open path to existing file
  7     SCRN  BLOCK                     \ address of source block
  8     16  0  DO
  9       ( adr )  DUP    64  I  *  +    64    TYPE
 10     LOOP
 11     >SCREEN ;                       \ return to normal output
 12 ->
 13 \ This command transfers a single screen.
 14 \ The command line has the form:
 15 \   scr# 0" file-name" XFR

SCREEN #6
  0 \ ADR  XFRS+                                            RDL012789
  1                                                        \
  2 VAR  ADR
  3                                                        \
  4 : XFRS+ ( first-scr# addr count - )                    \ RDL121988
  5     CR
  6     SWAP  TO  ADR                       \ save string address
  7     OVER  +  SWAP    DO                 \ set loop limits
  8       I  4  .R                          \ display source#
  9       I  ADR  XFR+                      \ transfer the block
 10     LOOP ;
 12 ->
 14 \ The command line has the form:
 15 \   <first-scr#> 0" <name>" count XFRS+

SCREEN #7
  0 : XFR   ( scr# adr - )                                 \ RDL012789
  1     SWAP  TO  SCRN
  2     ( adr )  >PATH                  \ open path to new file
  3     SCRN  BLOCK                     \ address of source block
  4     16  0  DO
  5       ( adr )  DUP    64  I  *  +    64    TYPE
  6     LOOP
  7     >SCREEN ;                       \ return to normal output
  8 ;S
  9 \ This command transfers a single screen.
 10 \ The command line has the form:
 11 \   scr# 0" file-name" XFR

+++
```



The Macintosh™ Section

Reserved as

A place for your thoughts

And ours.....

Mac-Watch

By James E. Law

Reviewing HyperDA and MultiClip

A Review of HyperDA

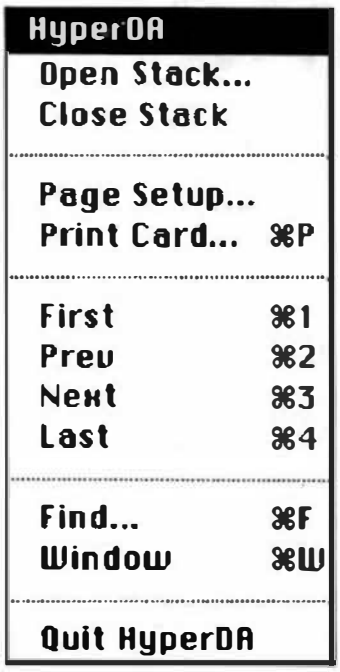
HyperCard has, without a doubt, represented a real step forward in making the power of the Macintosh available to application developers who do not have years of programming experience. The problem has been that HyperCard and its stacks are real RAM hogs. You can forget about running it on anything with less than a full MB of RAM. What may be worse, it cannot be effectively run under MultiFinder without at least 2 MB of RAM. During the RAM shortage, I'm sure many would be users of HyperCard were turned off by their inability to access information in its stacks without quitting the currently active application.

I am continually amazed at the ingenuity of Macintosh programmers, however, in finding effective work arounds for previously confining software of hardware limitations. In this case, Bill Appleton did what Apple should have done to start with. He developed a desk accessory which can be used while in any application to open and examine HyperCard stacks. It also allows you to copy text and graphic elements to the clipboard.

HyperDA by Symmetry Corporation is a 56K desk accessory and may be used on any Macintosh with at least 512K of RAM. It is easily installed in the Apple Menu just like other desk accessories. This product comes with a number of small HyperCard stacks including a well designed HyperDA Manual Stack.

Getting Started

When you open HyperDA, it places its own menu on the standard menu bar. You may then select OPEN STACK to see a list of HyperCard stacks. Select any of them and the stack is quickly opened. This menu bar also contains Close Stack, Page Setup, Print Card, Find, Quit HyperDA, selections for maneuvering through stacks (e.g., NEXT, FIRST, LAST).



The HyperDA Menu

Window Options

In HyperCard, the window usually takes over the entire screen. If HyperDA took the same approach, it would be difficult to easily switch between the open HyperCard Stack and other open documents or to view both simultaneously. HyperDA addresses this issue by allowing you to choose the format of the HyperDA window. You may elect to

view it as it would appear under the control of the HyperCard application or as a scroll able and re sizable window. The latter option works much better when you have other applications open concurrently with HyperDA.

What HyperDA Does

HyperDA allows you to do the sorts of things that you could do in HyperCards "browse" mode. You can view cards, progressing through the stack by clicking the appropriate buttons. Most (but not all) buttons may be activated by HyperDA. An extremely valuable feature is the ability to select and copy text and graphics from HyperCard stacks. When the cursor is moved over a text field, it turns into the typical "I" bar text tool and can be used to select and copy text. By holding down the OPTION key and dragging, graphic elements may be selected and copied.

Documentation

HyperDA is accompanied by a well designed and helpful manual. The HyperDA Manual stack contains essentially the same information as the hard copy manual and works very well. In practice, this product is so straight forward you probably will not need to refer to either manual.

Should You Buy it?

HyperDA does what it is suppose to do in a simple yet elegant manner. If you use HyperCard, you need HyperDA. This is especially true if you have less than two MB of RAM.

The original clipboard feature of the Macintosh presented new communication possibilities not readily available in other computers. The clipboard allowed transfer of text and graphics between applications and documents at will. The only problem was that the clipboard contained only one item. If you cut or copied an item into the clipboard, this item replaced what was already there. A number of times I have saved an item to the clipboard for later use and inadvertantly used the clipboard for another operation. The first item was then lost forever. So while the clipboard was a remarkable step forward for its day, its utility is limited by its capacity of only one image.

A Review of MultiClip

A Clipboard/scrapbook replacement that allows multiple clipboards.

Introducing MultiClip

It would be nice if Apple were more proactive in identifing and fixing limitations in its system software, but that doesn't seem to be the case. (Or will System 7.0 prove me wrong?) It's a good thing that the commercial spirit, or love of a good challenge, drives other talented programmers to provide enhanced system software. So it is with the clipboard's limitation. MultiClip by Olduvai Corporation is presented as a significant enhancement to the Apple clipboard. Let's see if it delivers.

MultiClip is an INIT and so is activated by placing it in your system folder and restarting your Macintosh. It requires at least 512Ke hardware and system software 6.0.2 or later.

Using MultiClip

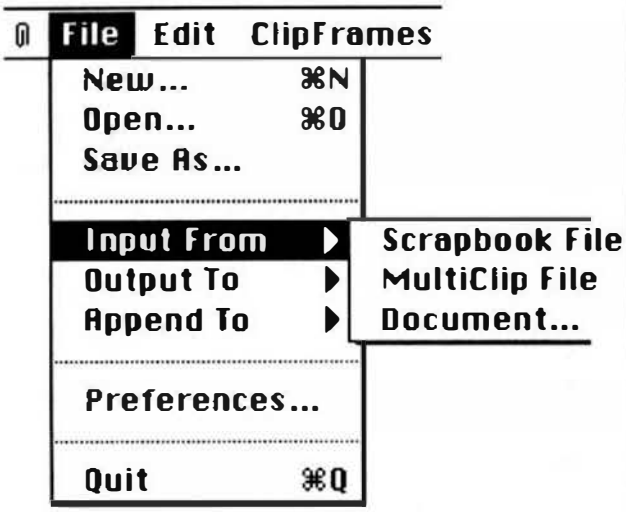
MultiClip can be used just like the old clipboard to facilitate cutting, copying, and pasting. The big difference is that you can cut or copy an almost unlimited number of items to the clipboard and all are preserved. When you want to paste from the Clipboard, you may paste sequentially in the same order that you placed the items into the clipboard. Alternately, you may elect to have MultiClip paste in the reserve order that the items were cut or copied. Obviously, there would be no practical value in retaining hundreds of past clipboard images. Usually you need to use only the last few. If you did maintain all those images, you would have no way to find the one you want. Consequently, MultiClip allows you to set a limit on the number of images to be maintained. When this number is reached, the oldest image is deleted.

MultiClip uses different key commands from the old clipboard (e.g., Command + Option +"X" to cut). This allows you to continue to use both clipboards if you wish (although I don't see an value in keeping the old one if you have MultiClip.

MultiClip has a sizable collection of "hot keys" that is, keyboard commands for speeding up the process of using the clipboard. The keys assigned to each function can be changed.

By pressing designated "hot keys," a window is displayed showing minature images of each item saved in MultiClip. This window is well designed to

give you good control of almost all activities involving the clipboards. By processing certain keys and dragging you may change the order of images in the MultiClip window and thus the order in which they are pasted back into your document. You may also duplicate and delete MultiClip images while viewing the MultiClip window. By double clicking an image, you can see it full-sized in a scrollable window.



Portion of Multi Clip Menu Showing Inport/Export Capability

Working with MultiClip Images

You can use a selection rectangle to select portions of graphic images in the MultiClip window and similarly can use a text tool to select text. Selected items can be copied or used to create a separate MultiClip frame. MultiClip doesn't stop there. When you view MultiClip frames containing text, you may edit that text (e.g., paste new text into the frame, cut text, and type in changes). When you compare this level of versatility and power with the old clipboard, you will see that MultiClip is light-years ahead.

MultiClip has powerful importing and exporting capability. By selecting "Input From" in the file menu, you can obtain and transfer into MultiClip images from scrapbook files, documents, or other MultiClip files. You may output to a scrap book file or to a document. When you output a graphic image to a document, you may elect to output it as a Paint or PICT image, either of which can then be opened and modified by the proper graphics application software.

Out With the Old; In With the New

During this review, I put MultiClip through its paces by using most of its features including a variety of imports and exports of files. Never once did it disappoint me. I like this program and suggest that you throw away the antiquated Apple clipboard DA and install a real work horse, that is, MultiClip.

By James E. Law
1806 Rock Bluff Rd.
Hixson TN 37343

Taming Software Timing Loops

By: Peter S. Gilmour

ABSTRACT

This article explains a technique of using the assembler's symbol manipulation power to control software timing loops by letting it calculate the instruction cycles used per loop and the required loop count constant. Assembler modifications to automatically support timing loop constant calculations are also presented.

About the author

Peter S. Gilmour is a Senior Systems Analyst with over 15 years experience. He is currently employed by the Motorola Microprocessor Group in Austin Texas where he works on the Motorola HDS-300 line of real time emulators for the Development Systems Group. He has a B.S. from Case Institute of Technology and a M.S. from Arizona State University. His interests include tennis, golf, and personal computing. He has several published articles, including a two part series titled Designer's Guide to fine-tuning uP/uC code in the May 25/June 8, 1989 issues of EDN, and Automated Software Testing in the April 6, 1989 issue of Machine Design. He can be reached at Motorola at (512) 891-2850.

Software timing loops have long been utilized to provide timing functions when no real time clock is available. The main use for timing loops is to control hardware functions when a fixed delay time is required between accessing two hardware functions to allow time for the hardware to respond, i.e. the hardware is "slow" compared to the speed of the processor. As an example, consider the Western Digital WD2793 floppy disk controller (FDC) chip. When a command is written to the chip's Command Register, a delay of at least 12 usec for double density, 1 MHz operation must be observed before the BUSY bit in thechip's Status Register becomes valid, i.e. the controllerneeds some time for "internal sync cycles" before it can set the BUSY bit. Another typical use for timing loops is a "timeout" function, i.e. a maximum amount of time is to be spent waiting for a certain condition before giving up. An example of a "timeout" function would be waiting a maximum of N msec for a character to arrive on a serial port before aborting. A non hardware use can be as simple as "wasting" some computer time so humans can read a display message before processing continues. The uses are endless, but managing, let alone taming these timing loops, has always seemed just beyond reach.

Previous methods to manage software timing loops have included empirical "trial-and-error" determination and hand calculations based upon clock frequency and instruction cycles used in the loop. The "trial-and-error" method usually involves additional equipment, such as oscilloscopes or frequency counters, to arrive at the right loop timing constant. It may even require special code segments to test the timing. Hand calculations are prone to errors and must be repeated for each instruction change in the loop. Unfortunately, both methods are not very well documented unless the programmer includes them as comments. There is no easy way to verify or modify a timing loop after it has been coded with these methods. Not waiting enough time causes unnecessary errors, including erratic operation or mysterious "crashes", while waiting too long causes a degradation in overall system performance.

A much simpler technique is to use the power of the assembler's symbol processing ("equate" and "set" statements such as found in Motorola's Macro Assemblers) to calculate timing constants at assembly time. By entering all parameters as symbols, the assembler can use its arithmetic capability to calculate the required timing constants. The two basic assembler symbol manipulation pseudo-instructions used to implement this technique are defined as follows:

EQU - assign the expression value to a symbol (cannot be redefined, i.e. it is "absolute")

SET - assign the expression value to a symbol (can be redefined)

When all parameters are defined as symbols, verification is reduced to checking the parameters versus the symbols and verifying the equations defining the timing constants. The use of symbols to enter the hardware timing requirements as read from specification sheets (in the stated time units) assures that the actual delay time will match the specification. Modifications are simplified by addition and/or subtraction of the required instructions and corresponding cycle count equations. Reassembly will then automatically recalculate the correct timing constant. This EQU/SET technique can be used in all designs when the CPU has a fixed execution cycle time, such as the Motorola M6800 family, M6805 family, MC6809, MC68HC11, MC68000, MC68008, and MC68010 (except Loop Mode Operation). CPUs with pipeline and/or instruction caching features, such as the Motorola MC68020 and MC68030, will not work.

The basic method utilized for all software timing loops is to execute enough instruction cycles at the known processor clock frequency to produce the required delay time. This translates to the simple formula of

T = X / F (Equation 1)

where:
T = delay time
X = total number of instruction cycles executed
F = clock frequency

Equation 1 can be broken down further as follows.

X = N * U (Equation 2)

where:
N = number of times thru the timing loop (timing constant)
U = number of instruction cycles per loop

Substituting Equation 2 into Equation 1 gives:

T = (N * U) / F (Equation 3)
Solving Equation 3 for N, the timing constant required for the delay loop yields:

T * F = N * U

N = (T * F) / U (Equation 4)

Because instruction cycles come in integer quantities, a "guard band" value must be entered to ensure the delay is "at least" the required amount of time. This avoids the truncation problem associated with integer arithmetic because fractional remainders are discarded. This is accomplished by modifying Equation 4 to add "U-1" to the numerator as follows so any fractional value will cause incrementing to the next integer value:

N = ((T * F) + (U - 1)) / U (Equation 5)

For a simplified example of the truncation problem, consider the following example.

Example 1.

Given: (T * F) = 8 cycles U = 3 cycles/loop

Equation 4 would yield:

N = 8 / 3 = 2

Thus only (N * U) = 6 cycles would be executed, not 8 as required.

Equation 5 would yield:

N = (8 + (3 - 1)) / 3 = 3

Thus (N * U) = 9 cycles would be executed, which is 1 more than the 8 required.

As can be seen, N = 3 is the minimum required value.

Since the clock frequency (F) and the required delay time (T) are known, only the number of instruction cycles executed per loop (U) is missing. By entering the number of cycles executed for each loop instruction immediately after each loop instruction, the assembler will "total up" the cycles executed per loop. Finally, a statement modeled on Equation 5 is constructed to calculate the required timing constant.

The only restrictions on using the assembler's symbol table processing are (1) integer arithmetic, (2) arithmetic overflow, and (3) round off errors. Thus the clock frequency can not be entered as 1500000 Hz, as it would certainly cause arithmetic overflow. Error conditions can be introduced by operand order, i.e. (A*B)/C may cause overflow, while (A/C)*B may cause a round off error, but (B/C)*A will yield the correct answer. To avoid these errors, great care and consideration for the order and value range of variables must be exercised when constructing the assembler timing equations. Also, be conscious of the symbol value storage size of the assembler (usually 16 or 32 bits), as this affects overflow during calculations

The EQU/SET technique can best be illustrated by examination of Listing 1.a. This listing shows a Motorola MC68HC11 processor timing loop that waits up to 1 msec for a bit in a hardware status register (memory location) to become set. The code exits when either the bit becomes set, or time has expired.

Line numbers have been added at the beginning of each line for identification purposes of the following discussion. Listing 1.a is an example of a software timing loop without using my technique. It is difficult to understand how long the timeout constant in line 40 causes the code to wait. Even if a comment was added stating the wait time, it isn't certain the program really waits that long, i.e. it would be difficult to verify the timing constant as correct.

Listing 1.a

```
030  STATUS  EQU     $FEC8    ; h/w status reg.: Bit7= 1 when ready
040          LDAB    #$46
060  LOOP    TST     STATUS
070          BMI     READY   ; Branch if status = ready!
080  DECB                     ; Decrement theloop count.
090  BNE     LOOP             ; Branch if not timed out yet!
100  BRA     EXPIRED ; Here for timeout!
110  READY   EQU     *        ; Here when ready!
```

Listing 1.b shows the same program but with lines added to use the EQU/SET technique. Line 10 defines the MPU clock frequency that this code will execute at, and line 20 defines the number of usec to wait for the status bit before timing out. Note that these values are entered in units of MHz and usec in order to simplify the equation calculation in line 92. Line 50 resets the instruction cycle count, U$, to zero for the timing loop which starts in line 60. For each instruction in the loop (lines 60, 70, 80, and 90), a line is inserted immediately afterwards (lines 61, 71, 81, 91) which adds the cycle count of the previous line to the previous cycle total, i.e. a running "subtotal". The instruction cycle count for each instruction is found in the M68HC11 Reference Manual, Appendix A, Instruction Set Details. For example, the manual shows that the TST instruction on line 60 uses 6 cycles because the addressing mode is "EXTENDED". The other cycle counts are obtained in a similar manner, based on opcode and addressing mode. Line 92 calculates the required delay count, DELCNT, using Equation 5 above. An equate (EQU) is used to define DELCNT because it is a fixed constant needed for this particular loop (used in assembler pass 2 to complete the instruction in line 40 which loads the timeout count). Other loops will have their own constants defined. In this example, the calculated cycle count, U$, is 0+6+3+2+3 = 14 ($0E) and delay count, DELCNT, is (1000+(14-1))/14 = 72 ($48).

Listing 1.b

```
010  FREQ    EQU     1        ; MPU clock frequency in MHz
020  RDYTIME EQU     1000     ; # of usec for status bit to be ready
030  STATUS  EQU     $FEC8    ; h/w status reg.Bit7= 1 when ready
040          LDAB    #DELCNT
050  U$      SET     0        ; Clear cycle count.
060  LOOP    TST     STATUS
061  U$      SET     U$+6
070          BMI     READY   ; Branch if status = ready!
071  U$      SET     U$+3
080          DECB             ; Decrement the loop count.
081  U$      SET     U$+2
```

```
090             BNE     LOOP    ; Branch if not timed out yet!
091   U$        SET     U$+3
092   DELCNT    EQU     ((FREQ*RDYTIME)+(U$-1))/U$
100             BRA     EXPIRED ; Here for time out!
110   READY     EQU     *       ; Here when ready!
```

If the need arose to insert an instruction in the loop, it would be done by inserting the lines below into their respective places.

```
075             NOP             ; Delay status accesses.
076   U$        SET     U$+2
```

Reassembly would change the calculated cycle count, U$, to 0+6+3+2+2+3 = 16 ($10) and delay count, DELCNT, to (1000+(16-1))/16 = 63 ($3F). Deleting an instruction is even easier; just remove the instruction line and its associated cycle calculation line, and reassemble.

Verification of the timing loop for Listing 1.b would involve checking each cycle count sub total line for the correct instruction cycles and checking the delay count calculation line for correctness.

Listing 2 shows a more advanced version of this technique to find the elapsed time for a quick RAM memory check from a bootstrap ROM using a Motorola MC68008 processor. The quick memory check is to be done while the user is reading the system signon/copyright message that has just been displayed on the user's terminal. This accomplishes two things, namely testing the RAM and delaying long enough so the signon message can be read. Thus one needs to know the amount of time consumed by the quick memory test. The system is supplied with either 512K or 896K bytes of RAM. The technique here is more complicated than that shown in Listing 1.b, because there are different numbers of wait cycles involved for read and write accesses to the different system resources, namely RAM, ROM, floppy disk controller (FDC), and Motorola MC68230 Parallel Interface/Timer (PI/T) chips. The code in Listing 2 executes in ROM (read) and accesses RAM (read and writes) in order to test it.

Lines 150-250 define the wait cycle counts for the various system resources during read and write accesses. Line 280 defines the clock frequency this code will execute at. Line 310 defines the test pattern value to be used. Lines 350-400 set up the registers outside the loop. Line 401 resets the instruction cycle count, U$, prior to entering the loop at line 420. The loop instruction lines (430, 450, 470, 490, 520, 540, 560, 580, 600, 620) are immediately followed by lines which total up the instruction cycles (431, 451, 471, 491, 521, 541, 561, 581, 601, 621). The instruction cycle count for each instruction is found in the M68000 8-/16-/32-Bit Microprocessors User's Manual. For example, the manual shows in Section 9, Table 9-4, that the MOVE.L instruction on line 430 uses 24 cycles for instruction fetch, 2 cycles for operand reads (from ROM) and 4 cycles for operand writes (to RAM). The read and write cycle times must then be multiplied by the number of wait cycles per read or write for the proper resource being accessed (ROM or RAM). The other cycle counts are obtained in a similar manner, based on opcode and addressing mode. Lines 631 and 632 calculate the number of usec to check 512K and 896K bytes of RAM respectively. In this example, the calculated values are $00390000 (3,735,552) for 512K and $0063C000 (6,537, 216) for 896K. Simple math converts these to 3.7 and 6.5 seconds, respectively. Executing the code verifies the calculations as correct.

Listing 2

```
010  * NOTES:
020  *       1.  System utilizes an MC68008 at 8  MHz clock rate with the following wait
030  *       cycles;
040  *       Wait Cycles/ Wait Cycles/
050  *Resource       Read Access     Write Access
060  *========       ==========      ==========
070  *RAM                    1               1
080  *ROM                    2               2
090  *FDC                    2               3
100  *68230                  3               4
110  *
120  *       Since this program runs in the ROM, the wait constants are defined as...
130  *       are defined as...
140  *
150  WC_R    EQU     1       ; wait cycles per      RAM     read  access
160  WC_W    EQU     1       ; wait cycles per      RAM write access
170  *
180  WC_RROM EQU     2       ; wait cycles per      ROM read  access
190  WC_WROM EQU     2       ; wait cycles per      ROM write access
200  *
210  WC_RFDC EQU     2       ; wait cycles per      FDC read  access
220  WC_WFDC EQU     3       ; wait cycles per      FDC write access
230  *
240  WC_R230 EQU     3       ; wait cycles per      68230 read  access
250  WC_W230 EQU     4       ; wait cycles per      68230 write access
260  *
270  *
280  FREQ    EQU     8       ; MPU clock frequency  in MHz
290  *  NOTE:16.000 MHz .01% crystal divided by 2 is used to obtain the 8 MHz
300  *       clock frequency!
310  PATTERN EQU $55555555          ; test pattern= every other bit ON
320  *
330  *  A1= calculated end of RAM address (512K or 896K)
340  *
350          SUBA.L  A0,A0   ; set A0.L= RAM test pointer
360  *                       ; (start at $0000)
370          MOVE.L  #PATTERN,D1     ; set D1.L= test pattern
380          MOVE.L  D1,D3   ; (takes fewer bytes than
390  *                       ;  another  MOVE.L #   instr!)
400  NOT.L   D3      ; set D3.L= inverse test pattern
401  U$      SET     0
410
420  RAMCHKLP:
430          MOVE.L  D1,(A0) ; store regular pattern
431  U$      SET
```

```
      U$+24+2*WC_RROM+4*WC_W
440
450           MOVE.L  (A0),D2 ; read back what was written
451   U$      SET     U$+24+2*WC_RROM+4*WC_R
460
470           CMP.L   D1,D2
471   U$      SET     U$+10+2*WC_RROM+0
480
490           BNE.B   RAMQER2 ; check regular pattern
491   U$      SET     U$+12+2*WC_RROM+0
500
510
520           MOVE.L  D3,(A0) ; store inverse pattern
521   U$      SET     U$+24+2*WC_RROM+4*WC_W
530
540           MOVE.L  (A0)+,D2; read back what was written
541   U$      SET     U$+24+2*WC_RROM+4*WC_R
550
560           CMP.L   D3,D2
561   U$      SET     U$+10+2*WC_RROM+0
570
580           BNE.B   RAMQER1 ; check inverse pattern
581   U$      SET     U$+12+2*WC_RROM+0
590
600           CMPA.L  A0,A1   ; check for end of RAM
601   U$      SET     U$+10+2*WC_RROM+0
610
620           BNE.B   RAMCHKLP; continue until end of RAM
621   U$      SET     U$+18+4*WC_RROM+0      ; # cycles for RAM test loop
630

631   time512 EQU     ($80000/4)*U$/FREQ     ; # usec to check 512K bytes
632   time896 EQU     ($E0000/4)*U$/FREQ     ; # usec to check 896K bytes
640
650
660   *** PASSED QUICK RAM TEST ***
```

An improvement over the EQU/SET method would be to modify the assembler to add the instruction cycle timing information and some features to utilize it, so the assembler will do all the work. The new features required are to add three reserved symbols (F$, C$, U$) and one pseudo-opcode (CALCDEL), as defined below. Reserved symbols are symbols exclusively assigned to the assembler for its use.

F$ absolute symbol whose value is the execution processor clock speed in MHz (real number).

C$ redefinable symbol whose value is the execution cycle count of the last instruction assembled.

U$ redefinable symbol whose value is the summation of the execution cycle counts of the previously assembled instructions, i.e. for each instruction,

U$ = U$ + C$.

CALCDEL assigns the value of the time delay equation (Equation 5) to the symbol in the label field.

Syntax:

<label> CALCDEL <delay_time>[,<time_units>] where <label> is the symbol name to receive the value of the time delay equation. <delay_time> is the minimum required delay time (T). <time_units> is the optional time units; USEC (default), MSEC, SEC, SECOND, SECONDS.

An assembler option could also be added which would include the instruction cycle count for each instruction (i.e. "C$") as part of the assembly listing. Using an assembler with these features, the program shown in Listing 1.a would be modified to that shown in Listing 3. Note how much more readable it is then the EQU/SET method shown in Listing 1.b. Real number capability allows for using the full range of processor clock speeds (Hz, KHz, MHz) and time delays (usec, msec, sec). To avoid truncation/overflow problems, CALCDEL should insert the calculated constant back into the time delay equation to ensure that the required delay is met. If it is not, increment the constant and repeat until the required delay is achieved.

Listing 3

```
010  F$       EQU     1.000   ; MPU clock frequency in MHz
020  RDYTIME  EQU     1000    ; # of usec for status bit to be ready
030  STATUS   EQU     $FEC8   ; h/w status reg.: Bit7= 1 when ready
040           LDAB    #DELCNT
050  U$       SET     0       ; Clear cycle count.
060  LOOP     TST     STATUS
070  BMI      READY   ; Branch if status = ready!
080  DECB             ; Decrement the loop count.
090  BNE      LOOP    ; Branch if not timed out yet!
092  DELCNT   CALCDELRDYTIME,USEC    ; Calculate delay constant
100  BRA      EXPIRED ; Here for timeout!
110  READY    EQU     *  ; Here when ready!
```

A further refinement would be to expand the symbol definition pseudo-opcodes (EQU, SET) to include a "units" value, such as time (USEC, MSEC, SEC) or frequency (HZ, KHZ, MHZ). This would allow the CALCDEL pseudo-opcode to automatically compensate for different "units" and it makes a much more understandable listing. For example, lines 10, 20, and 92 of Listing 3 would become as follows;

```
010   F$ EQU 1.000,MHZ; MPU clock frequency
020   RDYTIME EQU    1000,USEC; Time for status bit to be ready
092   DELCNT CALCDELRDYTIME; Calculate delay constant
```

Since the purpose of these delays is to ensure a minimum delay time is achieved, enabling interrupts in a system would only add the interrupt service time to the delay. Thus the technique will still funct on with interrupts.

In conclusion, it is possible to not only understand, but to control and maintain software timing loops by using the power of the assembler to calculate the required constants. By entering all timing variables as symbols, they become self-documenting. Adding and/or subtracting instructions is much easier, because reassembling will recalculate the required timing constants. Code verification is also much easier, because the task is broken into the smaller pieces of verifying symbol definitions, instruction cycle counts, and simple equation calculations.

References

MC68HC11 Reference Manual, M68HC11RM/AD, Prentice-Hall, Englewood Cliffs, N.J. 07632

M68000 8-/16-/32-Bit Microprocessors User's Manual, M68000UM/AD REV 5, sixth edition, Prentice-Hall, Englewood Cliffs, N.J. 07632

Western Digital 1983 Components Handbook. Western Digital Corp., 2445 McCabe Way, Irvine, CA 92714

PAT - JUST

PAT WITH 'C' Source $229.00
JUST WITH 'C' Source $79.95

OS-9 68K - 68008 - 68000 - 68010 - 68020 - OS-9 68K

PAT FROM S. E. MEDIA — A FULL FEATURED **SCREEN ORIENTED TEXT EDITOR** with all the best of PIE. For those who swore by and loved **PIE**, this is for **YOU! All PIE features & much more!** Too many features to list. And if you don't like ours, change or add your own. C source included. Easily configures to your CRT terminal, with special configuration section. No sweat!

COMBO
PAT
JUST
Special $249.00

JUST from S. E. MEDIA — Text formatter written by Ron Anderson; for dot matrix printers, provides many unique features. Output formatted to the display. User configurable for adapting to other printers. Comes set -up for Epson MX80 with Graflex. Up to 10 imbedded printer control commands. Compensates for double width printing. Includes normal line width, page numbering, margin, indent, paragraph, space, vertical skip lines, page length, centering, fill, justification, etc. Use with **PAT** or any other text editor. The ONLY stand alone text processor for the 68XXX OS-9, that we have seen. And at a very **LOW PRICE!** Order from: S. E. MEDIA - see catalog this issue.

MASTER CARD

VISA

Southeast East Media
5900 Cassandra Smith Rd
Hixson, Tn. 37343
Telephone (615) 842-4600
FAX (615) 842-7990

Shipping

U.S.A.	$4.50
CANADA	$7.50
FOREIGN	$25.00

THE 6800-6809 BOOKS

OS - 9 User Notes

By: Peter Dibble

The publishers of 68' Micro Journal are proud to make available the publication of Peter Dibbles

OS9 USER NOTES

Information for the BEGINNER to the PRO, Regular or CoCo OS9

Using OS9

HELP, HINTS, PROBLEMS, REVIEWS, SUGGESTIONS, COMPLAINTS, OS9 STANDARDS, Generating a New Bootstrap, Building anew System Disk, OS9 Users Group, etc.

Programming Languages

Assembly Language Programs and Interfacing; Basic09, C, Pascal, and Cobol reviews, programs, and uses; etc.

Disks Include

No typing all the Source Listings in. Source Code and, where applicable, assembled or compiled Operating Programs. The Source and the Discussions in the Columns can be used "as is", or as a "Starting Point for developing your OWN more powerful Programs. Programs sometimes use multiple Languages such as a short Assembly Language Routine for reading a Directory, which is then "piped" to a Basic09 Routine for output formatting, etc.

BOOK $9.95

Typeset — w/ Source Listings
(3-Hole Punched; 8 x 11)
Deluxe Binder $5.50

All Source Listings on Disk

1-8" SS, SD Disk **$14.95**
2-5" SS, SD Disks **$24.95**

FLEX USER NOTES

By: Ronald Anderson

The publishers of 68 MICRO JOURNAL are proud to make available the publication of Ron Anderson's **FLEX USER NOTES**, in book form. This popular monthly column has been a regular feature in 68' MICRO JOURNAL SINCE 1979. It has earned the respect of thousands of 68 MICRO JOURNAL readers over the years. In fact, Ron's column has been described as the 'Bible' for 68XX users, by some of the world's leading microprocessor professionals. The most needed and popular 68XX book available. Over the years Ron's column has been one of the most popular in 68 MICRO JOURNAL. And of course 68 MICRO JOURNAL is the most popular 68XX magazine published.

Listed below are a few of the **TEXT** files included in the book and on diskette. All TEXT files in the book are on the disks.

LOGO.C1	File load program to offset memory - ASM PIC
MEMOVES.C1	Memory move program - ASM PIC
DUMP.C1	Printer dump program - uses LOGO - ASM PIC
SUBTEST.C1	Simulation of 6800 code to 6809, show differences - ASM
TERMEM.C2	Modem input to disk (or other port input to disk) - ASM
M.C2	Output a file to modem (or another port) - ASM
PRINT.C3	Parallel (enhanced) printer driver - ASM
MODEM.C2	TTL output to CRT and modem (or other port) - ASM
SCIPKG.C1	Scientific math routines - PASCAL
U.C4	Mini-monitor, disk resident, many useful functions - ASM
PRINT.C4	Parallel printer driver, without PFLAG - ASM
SET.C5	Set printer modes - ASM
SETBAS1.C5	Set printer modes - A-BASIC

Note: .C1, .C2, etc.=Chapter 1, Chapter 2, etc.

** Over 30 TEXT files included is ASM (assembler)-PASCAL-PIC (position independent code) TSC BASIC-C, etc.

Book only: **$7.95** + **$2.50** S/H

With disk: 5" **$20.90** + **$2.50** S/H
With disk: 8" **$22.90** + **$2.50** S/H

Shipping & Handling $3.50 per Book, $2.50 per Disk set
Foreign Orders Add $4.50 Surface Mail
or &7.00 Air Mail
If paying by check - Please allow 4-6 weeks delivery
*** All Currency in U.S. Dollars**
Continually Updated in 68 Micro Journal Monthly

Computer Publishing Inc.
5900 Cassandra Smith Rd.
Hixson, TN 37343
Telephone (615) 842-4601
FAX (615) 842-7990

68' Micro Journal
Reader Service Disks

Disk- 1	Filesort, Miniicat, Miniscopy,Minilfms, **Lifetime, **Poetry, **Foodlist, **Diet.
Disk- 2	Diskedit w/ inst.& fixes, Prime, *Prinod, **Snoopy, **Football, **Hexpawn,**Lifetime.
Disk- 3	Cbug09, Sec1, Sec2, Find, Table2, Intext, Disk-exp, *Disksave.
Disk- 4	Mailing Program, *Finddat, *Change, *Testdisk.
Disk- 5	*DISKFIX 1, *DISKFIX 2, **LETTER, **LOVESIGN, **BLACKJAK, **BOWLING.
Disk-6	**Purchase Order, Index (Disk file indx).
Disk- 7	Linking Loader, Rload, Harkness.
Disk-8	Crtest, Lanpher (May 82).
Disk- 9	Datecopy, Diskfix9 (Aug 82).
Disk-10	Home Accounting (July 82).
Disk-11	Dissembler (June 84).
Disk-12	Modem68 (May 84).
Disk-13	*Initmf68, Testmf68,*Cleanup,*Dskalign,Help,Date.Txt.
Disk-14	*Init, *Test, *Terminal, *Find, *Diskedit, Init.Lib.
Disk-15	Modem9 + Updates (Dec. 84 Gilchrist) to Modem9 (April 84 Commo).
Disk-16	Copy.Txt, Copy.Doc, Cat.Txt, Cat.Doc.
Disk-17	Match Utility, RATBAS, A Basic Preprocessor.
Disk-18	Parse.Mod, Size.Cmd (Sept. 85 Armstrong),CMDCODE, CMD.Txt (Sept. 85 Spray).
Disk-19	Clock, Date, Copy, Cat, PDEL.Asm & Doc., Errors.Sys, Do, Log.Asm & Doc.
Disk-20	UNIX Like Tools (July & Sept. 85 Taylor & Gilchrist). Dragon.C, Grep.C, LS.C, FDUMP.C.
Disk-21	Utilities & Games - Date, Life, Madness, Touch, Goblin, Starshot, & 15 more.
Disk-22	Read CPM & Non-FLEX Disks. Fraser May 1984.
Disk-23	ISAM, Indexed Sequential file Accessing Methods, Condon Nov.85. Extensible Table Driven. Language Recognition Utility, Anderson Mar86.
Disk-24	68' Micro Journal Index of Articles & Bit Bucket Items from 1979 - 1985, John Current.
Disk-25	KERMIT for FLEX derived from the UNIX ver. Burg Feb. 1986. (2)-5" Disks or (1)-8" Disk.
Disk-26	Compacta UniBoard review, code & diagram, Burlison March '86.
Disk-27	ROTABIT.TXT, SUMSTEST.TXT, CONDATA.TXT, BADMEN.TXT.
Disk-28	CT-82 Emulator, bit mapped.
Disk-29	**Star Trek
Disk-30	Simple Winchester, Dec.'86 Green.
Disk-31	*** Read/Write MS/PC-DOS (SK*DOS)
Disk-32	Heir-UNIX Type upgrade - Feb .87
Disk-33	Build the GT-4 Terminal - Nov. 87 Joseph Condon.
Disk-34	FLEX 6809 Diagnostics, Disk Drive Test, ROM Test, RAM Test - Apr. 89Korpi.
Disk-35	DO A FLEX-09 Batch File Processor - Oct. 88 - Dave Howland
Disk-36	Add Graphics To Your SBC - Nov. 88 - Joseph Condon
Disk-37	Minix for the PT68K-2 - June/July 89 - J. GaryMills

NOTE:

This is a reader service ONLY! No Warranty is offered or implied, they are as received by 68' Micro Journal, and are for reader convenience ONLY (some MAY include fixes or patches). Also 6800 and 6809 programs are mixed, as each is fairly simple (mostly) to convert to the other. Software is available to cross-assemble all.

* Denotes 6800 - ** Denotes BASIC

*** Denotes 68000 - 6809 no indicator.

8" disk $19.50

5" disk $16.95

Shipping & Handling -U.S.A. Add: - $3.50

Overseas add: $4.50 Surface - $7.00 Airmail

68' MICRO JOURNAL

5900 Cassandra Smith Rd.

Hixson, TN 37343

(615) 842-4600

FAX (615) 842-7990

!!! Subscribe Now !!!

68 MICRO JOURNAL

Toll Free Subscription Line 1-800 669 6809

OK, PLEASE ENTER MY SUBSCRIPTION

Bill My: ☐ Mastercard ☐ VISA

Card #_____ Exp. Date_____

For 1 Year _____ 2 Years _____ 3 Years _____

Enclosed: $_____

Name _____

Street _____

City _____ **State** _____ **Zip** _____

Country _____

My Computer is: _____

My Operating System is: _____

Subscription Rates

U.S.A.: 1 Year $24.50, 2 Years $42.50, 3 Years $64.50

***Foreign Surface: Add $12.00 per Year to USA Price.**

***Foreign Airmail: Add $48.00 per Year to USA Price.**

***Canada & Mexico: Add $9.50 per Year to USA Price.**

***U.S. Currency Cash or Check Drawn on a USA Bank !**

68' Micro Journal

5900 Cassandra Smith Rd.
POB 849
Hixson, TN 37343

Toll Free Subscription Line
1-800 669 6809
FAX (615) 842-7990

PT-68000 SINGLE BOARD COMPUTER

The PT68K2 is Available in a Variety of Formats
From Basic Kits to Completely Assembled Systems

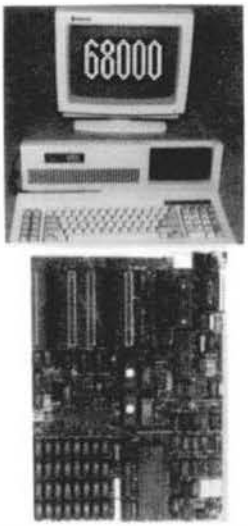
BASIC KIT (8 MHZ) - Board, 68000, HUMBUG MONITOR + BASIC in ROM, 4K STATIC RAM, 2 SERIAL PORTS, all Components **$200**

PACKAGE DEAL - Complete Kit with Board 68000 10 MHZ, SK*DOS, 512K RAM, and all Necessary Parts **$575**

ASSEMBLED BOARD (12 MHZ) Completely Tested, 1024K RAM, FLOPPY CONTROLLER, PIA, SK*DOS **$899**

ASSEMBLED SYSTEM - 10 MHZ BOARD, CABINET POWER SUPPLY, MONITOR + KEYBOARD, 80 TRACK FLOPPY DRIVE, CABLES **$1299**
For A 20 MEG DRIVE, CONTROLLER and CABLES **Add $295**

PROFESSIONAL OS9 **$500**



FEATURES

- MC68000 Processor, 8 MHZ Clock (optional 10,12.5 MHZ)
- 512K or 1024K of DRAM (no wait states)
- 4K of SPAM (6116)
- 32K,64K or 128K of EPROM
- Four RS-232 Serial Ports
- Floppy disk controller will control up to four 5 1/4", 40 or 80 track.
- Clock with on-board battery.
- 2 - 8 bit Parallel Ports
- Board can be mounted in an IBM type PC/XT cabinet and has a power connector to match the IBM type power supply.
- Expansion ports - 6 IBM PC/XT compatible I/O ports. The HUMBUG™ monitor supports monochrome and/or color adaptor cards and Western Digital winchester interface cards.

PERIPHERAL TECHNOLOGY

1710 Cumberland Point Dr., Suite 8
Marietta, Georgia 30067
404/984-0742
VISA/MASTERCARD/CHECK/C.O.D.

Send For Catalogue
For Complete Information On All Products

*SK*DOS is a Trademark of STAR-K SOFTWARE SYSTEMS CORP.
*OS9 is a Trademark of Microware

DATA-COMP SPECIAL

Heavy Duty Power Supplies

For A limited time our HEAVY DUTY SWITCHING POWER SUPPLY. These are BRAND NEW units. Note that these prices are less than 1/4 the normal price for these high quality units.

Make: Boschert
Size: 10.5 x 5 x 2.5 inches
Including heavy mounting bracket and heatsink.

Rating: in 110/220 volts ac (strap change) Out: 130 watts

Output: +5v - 10 amps
+12v - 4.0 amps
+12v - 2.0 amps
-12v - 0.5 amps

Mating Connector: Terminal strip
Load Reaction: Automatic short circuit recovery
SPECIAL: $59.95 each
2 or more $49.95 each
Add: $7.50 each S/H

Make: Boschert
Size: 10.75 x 6.2 x 2.25 inches

Rating: 110/220 ac (strap change) Out: 81 watts

Outputs: +5v - 8.0 amps
+12v - 2.4 amps
+12v - 2.4 amps
+12v - 2.1 amps
-12v - 0.4 amps

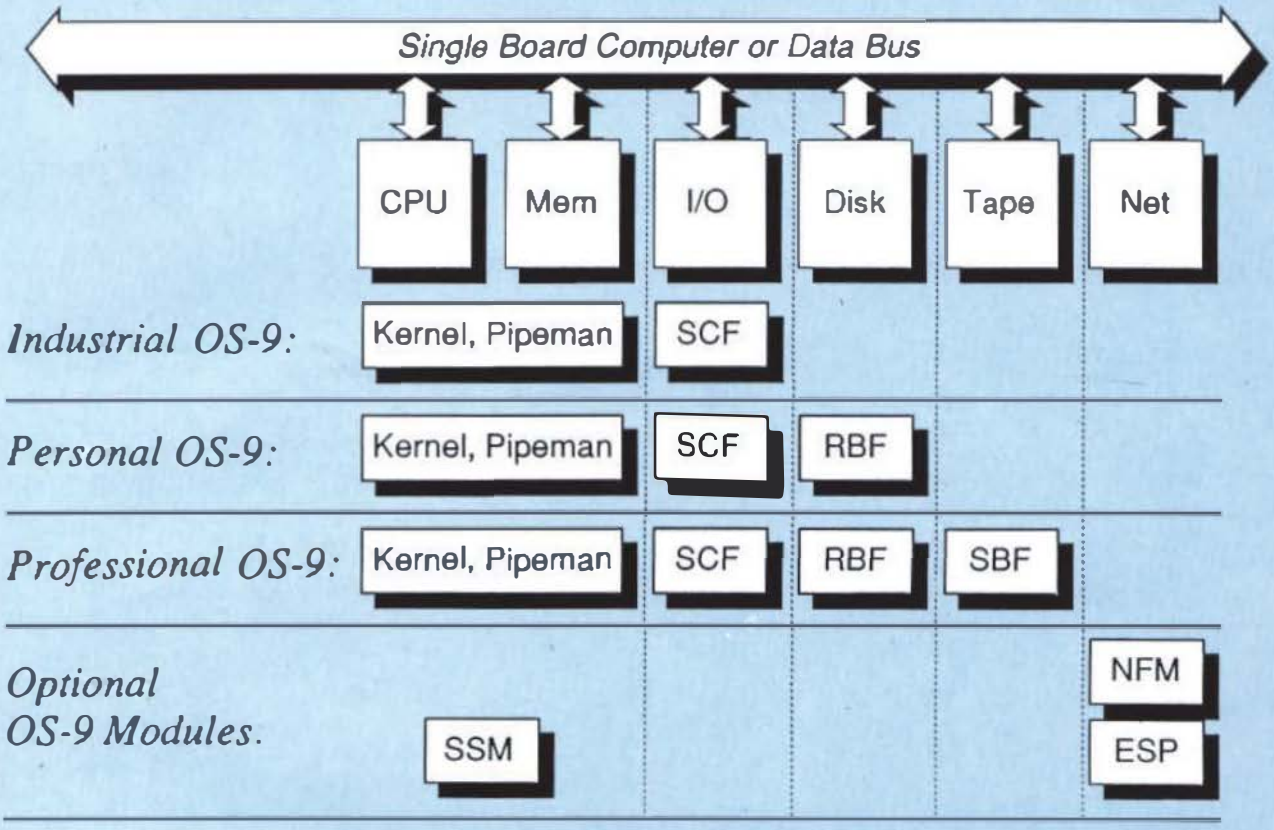
Mating Connectors: Molex
Load Reaction: Automatic short circuit recovery
SPECIAL: $49.95 each
2 or more $39.95 each
Add: $7.50 S/H each

5900 Cassandra Smith Rd., Hixson, Tn. 37343 Telephone 615 842-4600 Telex 510 600-6630 Fax (615)842-7990

Realtime & Multiuser/Multitasking

covers the whole spectrum for applications with the 680xx microprocessor family

OS-9



OS-9 is an approved, high performance operating system featuring a complete set of services and software:

- Languages: C, FORTRAN, PASCAL, BASIC, Modula-2, ADA*)
- Support Services: Technical Hotline Support worldwide, Compuserve, Training Seminars, User's Group
- Technical Services: Installation Support, Application Support, Customisation Services
- Software Development Support: Cross-Software Development Packages: Cross-Compilers, UNIBRIDGE-UNIX-Cross-Development Package
- Complete Application Software Packages: Networking, TCP/IP, Data Base Management Systems, Graphics Kernel System ....

*) Cross Compiler under VAX/VMS

OS-9 Features

- Real-Time Executive
- Multi-User, Multi-Tasking
- ROMable
- UNIX-like Shell Interface
- Interprocess Communications
- Modularity

OS-9 includes

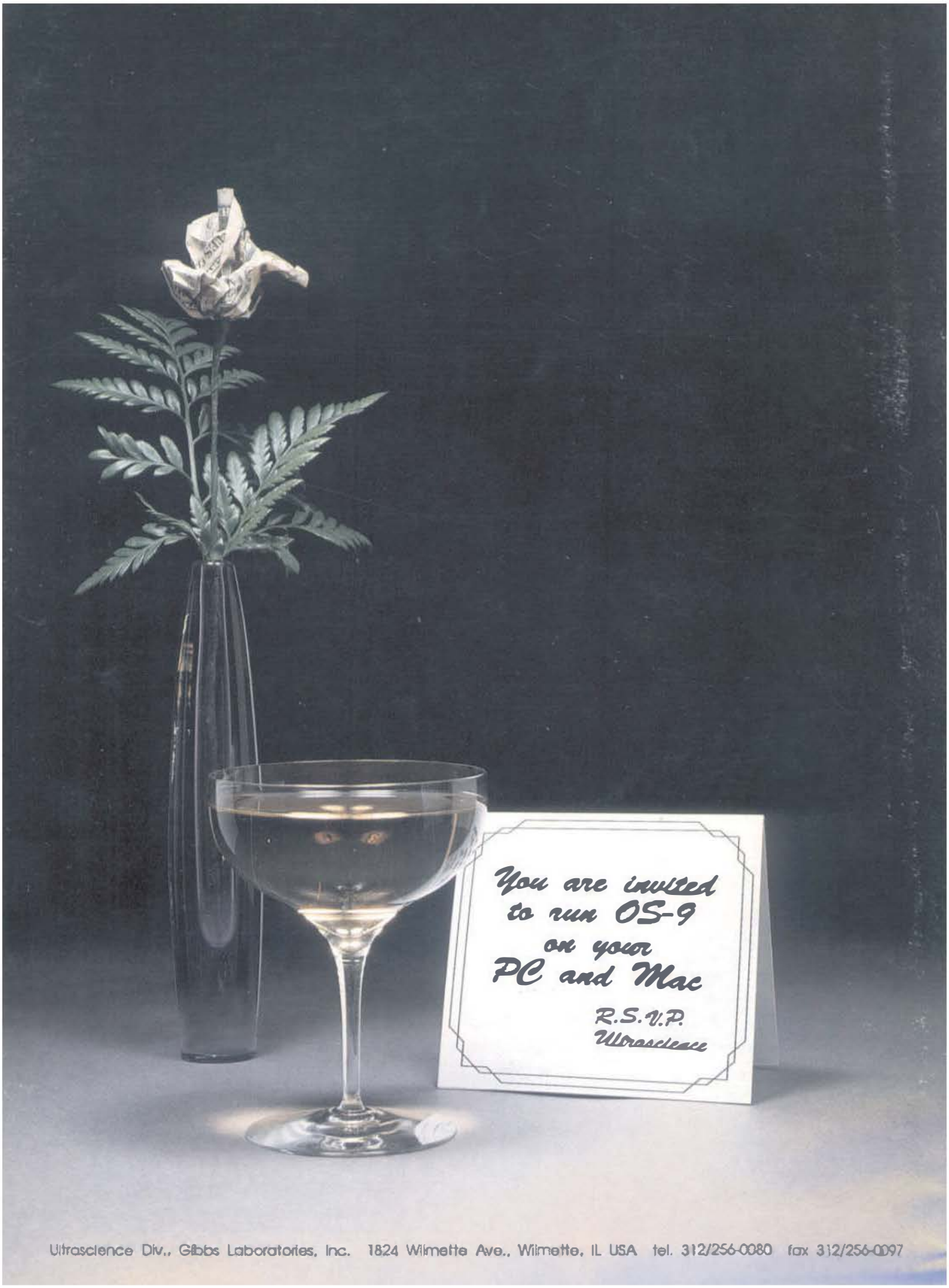
- Memory Management
- I/O Management
- File Management
- Process Management
- Complete Interrupt Services

microware®
authorized Distributor

DR. KEIL
Software·Elektronik·Datentechnik

Dr. Rudolf Keil GmbH
Gerhart-Hauptmann-Str. 30
D-6915 Dossenheim

Phone 06221/862091-93
Telex 461 166 keil d
Telefax 06221/861954



Ultrascience Div., Gibbs Laboratories, Inc. 1824 Wilmette Ave., Wilmette, IL USA tel. 312/256-0080 fax 312/256-0097